Softw Syst Model (2019) 18:445–471
https://doi.org/10.1007/s10270-017-0612-y

SPECIAL SECTION PAPER

Fast test suite-driven model-based fault localisation with application to pinpointing defects in student programs

Geoff Birch[1] · **Bernd Fischer**[2] · **Michael Poppleton**[1]

Received: 17 June 2016 / Revised: 21 March 2017 / Accepted: 13 July 2017 / Published online: 27 July 2017

Abstract Fault localisation, i.e. the identification of program locations that cause errors, takes significant effort and cost. We describe a fast model-based fault localisation algorithm that, given a test suite, uses symbolic execution methods to fully automatically identify a small subset of program locations where genuine program repairs exist. Our algorithm iterates over failing test cases and collects locations where an assignment change can repair exhibited faulty behaviour. Our main contribution is an improved search through the test suite, reducing the effort for the symbolic execution of the models and leading to speed-ups of more than two orders of magnitude over the previously published implementation by Griesmayer et al. We implemented our algorithm for C programs, using the KLEE symbolic execution engine, and demonstrate its effectiveness on the Siemens TCAS variants. Its performance is in line with recent alternative model-based fault localisation techniques, but narrows the location set further without rejecting any genuine repair locations where faults can be fixed by changing a single assignment. We also show how our tool can be used in an educational context to improve self-guided learning and accelerate assessment. We apply our algorithm to a large selection of actual student coursework submissions, providing precise localisation within a sub-second response time. We show this using small test suites, already provided in the coursework management system, and on expanded test suites, demonstrating the scalability of our approach. We also show compliance with test suites does not reliably grade a class of "almost-correct" submissions, which our tool highlights, as being close to the correct answer. Finally, we show an extension to our tool that extends our fast localisation results to a selection of student submissions that contain two faults.

Keywords Automated debugging · Model-based fault localisation · Symbolic execution · Automated assessment

Communicated by Prof. Alfonso Pierantonio, Jasmin Blanchette, Francis Bordeleau, Nikolai Kosmatov, Gabi Taentzer, and Manuel Wimmer.

✉ Geoff Birch
gb2g10@ecs.soton.ac.uk

Bernd Fischer
bfischer@cs.sun.ac.za

Michael Poppleton
mrp@ecs.soton.ac.uk

[1] University of Southampton, Southampton SO17 1BJ, UK

[2] Stellenbosch University, Matieland 7602, South Africa

1 Introduction

Fault localisation, i.e. the identification of program locations that can cause erroneous state transitions that eventually lead to observed program failures, is a critical component of the debugging cycle. Since it puts a significant time [47,50] and expertise burden [1,66] on programmers, a variety of different automated fault localisation methods have been proposed [12,14,23,25,26,34,55,56,58]. We describe a fast model-based fault localisation algorithm that, given a test suite, uses symbolic execution methods to fully automatically identify a small subset of program locations within which (under a single-fault assumption) a genuine program repair exists. Our main contribution is an improved search through the test suite that drastically reduces the effort for the symbolic execution of the models.

Model-based fault localisation [54] (sometimes also called model-based debugging [15]) is the application of model-based diagnosis methods [18] to programs. It involves three main steps: (i) the construction of a logical model from the

original program; (ii) the symbolic analysis of this model; and (iii) mapping any faults found in the model back to program locations. One popular approach to model-based fault localisation is to transform the program so that a symbolic program verification tool can be used for all three steps.

For example, Griesmayer et al. describe a method [23] in which the model (in the form of a logical satisfiability problem) is derived by running the CBMC model checker over a transformed program and then analysed by means of the model checker's integrated SAT solver. The transformation "inverts" the program's specification (cf. Sect. 2, producing failures where the original program would complete and blocking paths where the original program would fail), and replaces each original assignment by a conditional assignment with either the original value or an unconstrained symbolic value, depending on the value of a toggle variable. The actual localisation can then be reduced to extracting the possible values of the toggle variable from the satisfying assignments that the SAT solver returns.

However, the technique initially described by Griesmayer et al. requires detailed specifications to achieve acceptable precision—the weaker the specification, the more program locations are flagged as potential faults. Such detailed specifications rarely exist in practice. What *do* commonly exist, though, are extensive unit test suites, in particular in the context of modern test-driven design approaches. Griesmayer et al. have shown that their technique can (in principle) be extended to work with (failing) test cases, but the published results [23] are prohibitively slow, typically executing in hundreds to thousands of seconds per each small benchmark program. Griesmayer et al. have improved the original implementation [24] but only achieve times typically in the hundreds of seconds. Approximating model-based fault localisation approaches for test suites, such as Jose and Majumdar's method [34] (cf. Sect. 6.1), can run faster but can also miss a true fault location when evaluating a test case.

The bad run-time performance published by Griesmayer et al. is due to a naïve search algorithm that simply iterates over all test cases and runs an unoptimised "full width" search over all possible locations for each test case (cf. Sect. 2 for more details). However, the more locations the solver needs to explore, the longer the analysis of each test case takes. Moreover, the algorithm contains no optimisations to deal with test cases that generate intractable problems. We reimplemented this technique using modern hardware and a recent model checker on top of a state-of-the-art SMT solver in order to evaluate the current performance issues of this design. This reimplementation, while faster, confirms that the problems are caused by the naïve search.

We have further developed, implemented, and evaluated a novel approach that addresses these shortcomings that leads to typical speed-ups of more than two orders of magnitude over Griesmayer's results and yields a performance in line with current approximation techniques such as Jose and Majumdar's method [34]. Our approach still iterates over the failing test cases and runs a Griesmayer-style localisation task for each individual test case, but it maintains a *whitelist* of still viable fault locations which is narrowed down as the localisation tasks return. Our algorithm manages individual localisation tasks via a task pool to take advantage of the underlying multi-core hardware of modern systems, and dispatches the tasks in batches to percolate improvements in whitelist narrowing generationally. Tasks that fail to complete in a dynamically adjusted time are terminated and, if the whitelist is smaller, resubmitted at the tail of the iteration, where they may have become tractable due to the reduced search space leading to a smaller SMT problem for the solver. This early termination and resubmission approach increases the speed with which we can process larger test suites, without harming the localisation performance, as they typically have more redundant (for localisation purposes) test cases. It also prevents a loss of completeness in the results that model-based approaches can provide, within the limits of the symbolic analyser's accuracy.

Our approach is compatible with the modern test-driven design approach of specification by unit test suites. It inherits the typical strength of fault localisation techniques based on dynamic analysis, in that no prior knowledge of the program under test is required beyond test cases being flagged as passing or failing. We implemented our algorithm in a tool with the use of ESBMC [17] and KLEE [10,11] symbolic analysers and with PyCParser [3] handling the program transformations to encode the specification and other model constraints. The algorithm inherits the underlying behaviour, in respect of library and method calls, unrolling loops, and so on, of the symbolic analyser used. We initially demonstrate [5] this algorithm on the defective TCAS program variants from the Siemens [29] repository, a set of common localisation benchmark programs. TCAS is a loop-free integer program with around 40 assignments over 170 lines of C code.

We have then applied our tool to a corpus of student programs [6]. This application expands from a branching, loop-free integer benchmark program to include student submissions that add coverage of programs using loops, although limitations of our symbolic analyser prevent analysis of programs using floating-point values. Our translation of these programs from Python and Java to C removed any programs that made use of standard library calls, as these are not compatible between languages. Our experiments with this real-world student code are encouraging. The short list of locations where an assignment repair is possible, provided by our tool, can aid students in pinpointing improvement sites before they resubmit their code. This should assist students in surmounting the final hurdle to completing writing of source

code that fully complies with a test suite without requiring advanced debugging skills. The run-times of our tool are competitive with common, fast spectrum-based localisation techniques while providing more accurate fault localisation. This is of value to novice programmers, who stop debugging when provided with large lists of potential repair sites. Sub-second processing times and use of test suites for specification allows our tool to work with existing coursework submission systems and workflows, both for self-training and grading, in principle even at the scale required by Massive Open Online Courses (MOOCs).

Our tool can also provide instructors with automated assistance detecting some submissions that would provide perfect compliance after a single assignment edit (e.g. only fail due to a small mistake such as failing to increment a counter before returning it as the final value) but potentially fail most or all of a test suite. This will allow test suite-based automated graders to identify submissions whose quality is being underestimated by this functional assessment method. We show that real-world student submissions that are a single assignment edit away from full compliance with a test suite are not provided with a fair mark by grading against that test suite. Repair location information provided by our tool can also help instructors who are manually grading submissions to more quickly find and understand the mistakes made by students.

This article extends our paper [5], which proposed the algorithm described in Sect. 3, and that is shown operating on the TCAS program variants in Sect. 4. We present a new, more comprehensive example to illustrate our transformations and extend the analysis to contrast our performance with that of spectrum-based fault localisation techniques. We include a new analysis in Sect. 5 of the narrowing performance of our tool as applied to part of a corpus of actual student programs first introduced in our paper [6]. We introduce an extension to our tool that applies to dual-fault programs, with preliminary results on another part of the corpus of student program submissions. We conclude this article with an exploration of the related work in both fault localisation and student code analysis before providing concluding remarks.

2 Model-based fault localisation

Model-based fault localisation techniques are derived from the extension of model-based diagnosis [18,54] to the software domain.

A common process for model-based fault localisation uses static analysis on a model that is generated from a transformed input program using the language specifications. This transformation is provided by a checker tool that converts the program code into one or more logic expressions, and so analyses the symbolic execution of the system. A solver is

```
1  int payMe(uint nH, uint otH, uint age) {
2    uint BR = 15;
3    uint bonus = BR * (otH * 2);
4    uint normalPay = BR * nH;
5    uint ageHigher = age % 20;
6    uint SelBR = BR + ageHigher;
7    uint SelBonus = otH * (3 * SelBR);
8    uint SpecialNormalPay = nH * SelBR;
9    uint specialPay = BR + 20;
10   if ((age > 20) && (age < 40)) {
11     return SpecialNormalPay + SelBonus;
12   } else if (age >= 40) {
13     return (nH * specialPay) + ((specialPay * 2)
           * otH);
14   } else {
15     return normalPay + bonus;
16   }
17 }
18
19 int main(int argc, char ** argv) {
20   int result = payMe(atoi(argv[0]), atoi(argv[1])
       , atoi(argv[2]));
21   assert(result == atoi(argv[3]));
22 }
```

Fig. 1 Worked transformation: code with test case specification

invoked that can evaluate the logic expressions. This symbolic analysis may be accelerated with the use of built-in theories to compactly reason over the logic expressions [46]. This returns constraint satisfiability results for the expressions. The symbolic analyser uses these results to generate traces of specification violating execution paths (or *counterexamples*), if any exist. Some methods operate directly on the satisfiability result for the logic expression, for example exploring common omissions when using a maximum satisfiability solver [34]. Programs can be modified, e.g. augmented with additional predicates, to generate more information from the data in the failing traces.

Griesmayer et al. described a technique [23] where a specified input C program is reconfigured to use an "inverted" version of the specification. This inversion is applied to the C source code before the model is generated by, in their implementation, the CBMC model checker, which then analyses the model using a SAT solver. The reconfiguration generates a model based on a single-fault assumption.

Our worked example is based on the form of a student coursework submission used in the data set described in Sect. 5.1. The listing in Fig. 1 is taken after the program specification has already been encoded into the main procedure of the coursework submission. As detailed in Sect. 3.1, this is not part of the automated technique described by Griesmayer et al. This program calculates the pay owed to various employees based on the formulation given in a written question, as follows:

> "The pay that an employee earns each week depends on their age, the number of hours worked during normal business hours, and the number of overtime hours worked. The base pay rate for all workers is $15 per hour. On top of the base rate, each worker over the age of 20 earns an extra $1 per hour for every year their age

```
1  int payMe(uint nH, uint otH, uint age) {
2    uint __t = sym();
3    assert(__t < 11);
4    uint BR = (__t==0)? sym() : 15;
5    uint bonus = (__t==1)? sym() : BR*(otH*2);
6    uint normalPay = (__t==2)? sym() : BR*nH;
7    uint ageHigher = (__t==3)? sym() : age%20;
8    uint SelBR = (__t==4)? sym() : BR+ageHigher;
9    uint SelBonus = (__t==5)? sym():otH*(3*SelBR);
10   uint SpecialNormalPay= (__t==6)?sym():nH*SelBR;
11   uint specialPay = (__t==7)? sym() : BR+20;
12   if ((age > 20) && (age < 40)) {
13     return (__t==8)? sym() : SpecialNormalPay+
           SelBonus;
14   } else if (age >= 40) {
15     return (__t==9)? sym() : (nH*specialPay)+((
           specialPay*2)*otH);
16   } else {
17     return (__t==10)? sym() : normalPay+bonus;
18   }
19 }
20
21 int main(int argc, char ** argv) {
22   int result = payMe(atoi(argv[0]), atoi(argv[1])
         , atoi(argv[2]));
23   assert(result == atoi(argv[3]));
24 }
```

Fig. 2 Worked transformation: model of specified code

```
1  int payMe(uint nH, uint otH, uint age) {
2    uint __t = sym();
3    assume(__t < 11);
4    uint BR = (__t==0)? sym() : 15;
5    uint bonus = (__t==1)? sym() : BR*(otH*2);
6    uint normalPay = (__t==2)? sym() : BR*nH;
7    uint ageHigher = (__t==3)? sym() : age%20;
8    uint SelBR = (__t==4)? sym() : BR+ageHigher;
9    uint SelBonus = (__t==5)? sym():otH*(3*SelBR);
10   uint SpecialNormalPay= (__t==6)?sym():nH*SelBR;
11   uint specialPay = (__t==7)? sym() : BR+20;
12   if ((age > 20) && (age < 40)) {
13     return (__t==8)? sym() : SpecialNormalPay+
           SelBonus;
14   } else if (age >= 40) {
15     return (__t==9)? sym() : (nH*specialPay)+((
           specialPay*2)*otH);
16   } else {
17     return (__t==10)? sym() : normalPay+bonus;
18   }
19 }
20
21 int main(int argc, char ** argv) {
22   int result = payMe(atoi(argv[0]), atoi(argv[1])
         , atoi(argv[2]));
23   assume(result == atoi(argv[3]));
24   assert(0);
25 }
```

Fig. 3 Worked transformation: inverted model of specified code

> exceeds 20. However this additional, age-based bonus is only valid up until the age of 40. Finally, any overtime hours are paid at twice the base rate. Write a function which calculates the amount paid to an employee in one week, based on the number of normal and overtime hours worked, as well as their age. You should work in whole numbers (integers) only."[1]

The failing program, at the sixth assignment (line 7), declares an unsigned integer `SelBonus` equal to `otH*3*SelBR`. This expression contains a fault as the question text requires the bonus rate be double pay, not triple pay. This fault can surface, for some inputs, in the final output as a failure when returned in line 11 to `result` in line 20. Each failing test case in the program's test suite is taken as a search branch to be explored for potentially faulty assignment locations. To enable the search for such potential fault locations, an `unsigned int` global toggle variable, `__t`, is added that allows any one location to run alternative code (cf. Fig. 2). In the program body this toggle is made symbolic and constrained to the range of locations being explored (lines 2 and 3). The postcondition is encoded as an `assert` statement that will be transformed to an `assume` in the next stage of the process. Assignments in the source program are modified to become conditional assignments that flip to an unconstrained symbolic value generated by a call to `sym()`, when toggled. The single global toggle variable creates a model with a single-fault assumption.

Finally, this model is inverted which forces the verifier to suppress any assertion failures in the original model but to generate new counterexamples if the program can terminate normally without violating these assertions. The generated counterexamples from this new inverted model provide the toggle values that identify candidate assignment locations as sites of possible repair.

Each potentially failing `assert`-statement is replaced by a blocking `assume`-statement with the same argument. These `assumes` constrain the symbolic analyser to only generate counterexamples that violate any new `asserts` inserted into the program if the path did not violate the transformed `asserts` when encountered as `assumes`. Failing `assert(0)` statements are inserted before the program's terminal nodes to force the generation of these new counterexamples. The purpose of this inversion, shown in Fig. 3, is to provide the inverted output. Hence, traces that originally returned counterexamples due to specification failure do no longer, and traces that satisfy the specification now generate counterexamples. The exploration becomes one searching for a trace to the exit point that satisfies the initial specifications. Note that if the inversion was applied without the added non-determinism from the toggled assignments added in Fig. 2, no exiting trace would be found for any failing test case, because the specification, which is now encoded as `assume`-statements, blocks paths reaching the terminal `assert(0)`s.

Loop handling of the program under test is pushed to the symbolic analyser. Bounded model checking will unfold the program with unwinding of the loops to a bound of k and inlining function calls. Recursive calls will also be inlined, up to a bound. This explores a subset of all traces of the program up to the bound of the unfolding. The program transformation needs to take into account how the symbolic analyser handles

[1] Taken from http://www.dreamincode.net/forums/topic/288718-possible-math-error/.

loops. When a fault exists within a loop then the discussed configuration of symbolic expression insertion will replace every instance of the assignment with a new call to generate a symbolic value, allowing a different symbolic value to be chosen for each instance of the assignment. When that assignment is activated by the toggle value, all instances of the symbolic substitution will be triggered during that path trace. In other words, we localise a fault within a loop independent of the precise loop iteration or iterations in which it occurs. This adds to the solver complexity of any fault in a loop (or function called multiple times) because the solver needs to find the correct values (which are already computed by the program in the correct iterations) to assign the symbolic expressions. However, this approach is necessary to allow the search to isolate such faults to the same standard as other locations searched.

The architecture of our approach constrains the scope of programs where a repair can be localised to that of the underlying execution engine. KLEE has been shown to operate with symbolic values on C programs up to ten thousand executable lines of code long [11, Figure 5, p. 216]. Scalability of solver-based engines is difficult to predict. When KLEE is used, the behaviour with respect to inlining of external functions when source code is not available is to concretely execute the library call and use the returned value. This is not possible for symbolic arguments and so constrains which programs can be handled. The version of KLEE used in these results relies on a version of the STP solver that lacks symbolic floating-point handling. This limitation has since been removed and assignments using floating-point values will be localisable in future revisions.

This complete transformation effectively searches the model to find assignments that are possible repair locations. That is, every such assignment can be edited to a symbolic value that corrects the flow of *one* failing test case and results in the desired output being generated, i.e. not violating the `assume(result==atoi(argv[3]))` statement in the worked example's inverted model. Assignments where a symbolic value is found for *every* failing test case are returned by our approach as repair locations.

The localisation process effectively generates a lookup table at each returned location that will repair all failing test cases. For each failing test case, the alternative value of the assignments will be reported (via a counterexample) for each location flagged by this process. The flagging process means the chosen assignment values lead to the end of the program without failure of the original specification. All passing test cases define their own correct values for the assignments (they already execute within the test suite specification). So this lookup table is a genuine repair for the full test suite, albeit repairing only from the test cases provided as specification.

Unlike most other localisation processes, which only aim to isolate locations as suspicious, this process generates artefacts that define a repair based on values from a lookup table for the specification, as given by the full test suite. The repair is not one that a programmer would accept as complete. But it is a repair that, at minimum, provides significant additional debugging information in the form of what values the final repair can output to fix the faulty assignment under the specification of the test suite.

Using test suites demonstrates a strength of dynamic analysis: no prior knowledge of the program beyond flagged test cases as correct or incorrect is required. But the approach originally described by Griesmayer et al. treats each test case as an independent specification and collects all results independently. This results in prohibitively slow execution time and, in fact, the published results indicated that the run-time cost was too high using the common localisation performance measure of the Siemens TCAS [29] variants. We modify this approach to make the search of failing test cases dependent on the discovered negative search results, maintaining a whitelist of still viable fault locations which is narrowed down as the localisation tasks return.

To implement this extension of the above process, we use a design that takes advantage of modern consumer processor architectures to radically reduce wall-clock execution time [5]. Each test case is dispatched as a task to a worker pool with any pruning percolated to future tasks. We minimise the symbolic execution load with two features. Firstly, we intelligently prune the search space, i.e. we do not search known-unrepairable assignments in subsequent test cases after the search has started. Secondly, we minimise the disruption of an intractable (a risk of model checking programs) or slow search branch by monitoring and evicting tasks, which are retried later, once the search space has been pruned.

3 The algorithm

We propose a fast algorithm that optimises the search of the test suite and viable repair locations to bring the process into line with current performance expectations and without compromising the completeness of the results this technique allows. Each test case in the suite comprises an input string, which becomes the argument vector, and a desired output string from an oracle version of the program variant. We discuss the algorithm design with respect to the C programming language, but it can be applied to any language with suitable support for symbolic analysers.

We transform an input C program using an extended Griesmayer inversion process detailed in Sect. 3.1. This is handed to a worker task (see Sect. 3.2) with a whitelist of locations to search and a single failing test case. The returned set from the worker is a narrowed whitelist of locations that the test case

flagged as being potential repairs. We manage the process using the main tool loop (see Sect. 3.3). In the algorithms below we denote C programs as C and (after transformation) D and E. Individual failing test cases are f from the non-empty queue F. The refining whitelist of locations is L, which is narrowed during the process shown in Fig. 5 to generate the smaller whitelist K as output. In the Process Manager algorithm, P is the worker pool manager.

We provide two main contributions with this algorithm design. First, the reduction in symbolic execution work via the use of a narrowing process of whitelisting locations searched. Second, the management of cases where the time to return a narrowed whitelist is significantly beyond the mean time for a failing test case. This later case can be due to either an intractable model representation or poor narrowing compared to other unexplored failing test cases. The management is designed to provide a more consistent completion time over a range of different localisation searches. This aims to avoid slowdown from the highest cost branches of the search rather than focussing on providing an optimal search when all branches are cheap to search.

3.1 Program transformation into model

The program under analysis is transformed in two stages. First, an initial generic transformation *ApplyGenericTransforms* is applied once, as described in Sect. 3.3. This includes the Griesmayer inversion, as outlined in Sect. 2, and some accommodation of test case input and output data as inline specification. In the second stage *ApplySpecificTransforms* is applied for each subsequent test case processed. This implements any whitelist narrowing possible at this iteration via the toggle; for ESBMC it also includes some test case-specific input data encoding into the program text.

When we introduced our worked example in Sect. 2, Fig. 1 was already specified. A more faithful form of the student coursework submission on which it is based is given in Fig. 4. Note that this is missing the assertion on the result (line 21, Fig. 1) and uses C preprocessor constructs like the definition in line 1. This is the source that our tool must be able to work with to provide a fully automated tool-chain.

The previously discussed Griesmayer inversion (Fig. 3) is always applied during the generic stage, *ApplyGenericTransforms*. Also at this stage, we automatically encode the test case specification into the input program, avoiding the need to manually hard-code the inputs into the program. The input and output of C programs are defined by the passed program arguments, `argv`, and the standard inputs and standard outputs. For the fully automated execution of our tool, the input and desired output are encoded into the program via an expanded standard input; we widen `argv` to also accept the desired output as an extra string value. This can be compared in the transformed program against modified stdout

```
1  #DEFINE BR 15
2
3  //normalHours, overtimeHours, age
4  int payMe(uint nH, uint otH, uint age) {
5    uint bonus = BR * (otH * 2);
6    uint normalPay = BR * nH;
7    uint ageHigher = age % 20;
8    uint SelBR = BR + ageHigher;
9    uint SelBonus = otH * (3 * SelBR);
10   uint SpecialNormalPay = nH * SelBR;
11   uint specialPay = BR + 20;
12   if ((age > 20) && (age < 40)) {
13     return SpecialNormalPay + SelBonus;
14   } else if (age >= 40) {
15     return (nH * specialPay) + ((specialPay * 2)
         * otH);
16   } else {
17     return normalPay + bonus;
18   }
19 }
20
21 int main(int argc, char ** argv) {
22   int result = payMe(atoi(argv[0]), atoi(argv[1])
       , atoi(argv[2]));
23 }
```

Fig. 4 Original faulty code

commands, replacing calls to e.g. `printf` with comparisons that increment a pointer to the desired output when it matches the previous output of the `printf`. Assertions inserted before the program's terminal nodes, which confirm the pointer to the desired output has reached the end of the string, will complete this encoding of the specification. This is used in the generic implementation of automated postcondition insertion that our tool defaults to. It can be replaced when given a template that, for example, directly asserts the desired output matches the final variable holding the result of the program.

When only a single integer value is required to be checked for output specification compliance, such as in TCAS and our worked example, we can skip this string traversal. Rather than encoding the output as a string that is walked, it can be handled in the same way most input integers are. A call to `printf("%d", val);` can be replaced with `assert(val == atoi(argv[X]));` for X being the last value of the now-widened input vector. A program that returns a value to the calling program or operating environment will expose `return` calls in `main` that can be prepended with a similar assertion. In our worked example from the coursework database, the test cases define the name of the variable that must store the output value. In line 22 of Fig. 4 the value of `result` is tested by the inserted `assert(result == atoi(argv[3]));` in Fig. 1.

Program transformations can extend the number of assignments visible for repair. For example, a return statement that calculates a result before returning it can be divided into two steps with a temporary variable. Thus, `return a+b*c` becomes `temp=a+b*c; return temp`, exposing the expression to an assignment-based repair. If returns are always considered to be implicit assignments, then local-

isation is expanded from assignment locations to also include all return statements. In lines 13, 15, and 17 of Fig. 2, this can be seen applied to expose the implicit assignments of the expressions returned without needing to make the temporary assignment explicit.

In line 1 of Fig. 4, there is a statement `DEFINE BR 15`. This would be automatically integrated into the C program by the CPP preprocessor via token substitution, where it could not be reasoned on by the symbolic analyser as a single assigned value. Our tool checks for all instances of the token and embeds a variable that matches the required scope, as seen in line 2 of Fig. 1. This extension of the transformation process allows the symbolic analysis of this value to test it as a site of potential repair and can be mapped back to the original `DEFINE` statement by the reporting tool.

During the specific stage, i.e. *ApplySpecificTransforms*, the whitelist of locations must be applied to narrow the range of toggles being searched. We do this by adding assumptions of the form `assume(__t != 6);` immediately after `__t` is assigned a symbolic value, blocking searching the toggle point activated by this assignment.[2]

For ESBMC there is no way to populate `argv` in the program simulation, so the `argv` values need to be hard-coded into the transformed program before handing to the symbolic analyser. As this is linked to the specific test case, this can only be done at the specific stage. In our worked example, lines 22 and 23 of Fig. 3 would be modified during this transform pass to swap the `atoi(argv[X])` calls with the values from the test case being tested. KLEE provides a POSIX implementation for program I/O during simulation. This allows the test case input to be fed into the standard `argv` parameters and removes the requirement for this specific stage transform, Fig. 3 would be used as written.

3.2 The test case search algorithm

The algorithm in Fig. 5 outlines a single task that defaults to being deployed to a single core via the pool manager discussed in Sect. 3.3. When *AddTask* is called in lines 5 and 24 of Fig. 6 by the manager then an instance of this single unit of work is queued into the worker pool. These tasks run independently until they return their narrowed list of locations, K, in line 9 or are ejected by the pool manager. Each task takes an input program that has already been transformed by the generic stage, a single failing test case, and a whitelist of locations that are indicated by their associated toggle values.

ApplySpecificTransforms is discussed in Sect. 3.1. The final transformed program is generated, ready for submission to the symbolic analyser. The toggle values are restricted to the whitelist and, in the case of ESBMC, the test case is hard-coded into the source code.

CallModelCheckerOnInput passes the transformed program to the model checker. For ESBMC the failing test case has been encoded into the source code (in *ApplySpecificTransforms*), but KLEE still requires this information. KLEE, using the POSIX runtime environment model, is passed the argument vector (extended to contain the desired output) during the simulation of the program execution.

The call to *CallModelCheckerOnInput* returns the counter-examples for this failing test case, which are a set of traces that result in the raising of specification failure when simulating the execution of the transformed program. The traces are parsed into a format that holds the failure type and the associ-

[2] This implementation keeps the compact representation of the toggle values without requiring a full transformation pass to rewrite all ternary expressions.

```
Input: Program D; Failing Test Case f; Location Set L
Output: Fault Location Set K
 1: E = ApplySpecificTransforms(D, L, f);
 2: CounterExamples = CallModelCheckerOnInput(E, f)
 3: K = [];
 4: for c in CounterExamples do
 5:    if c.UnconditionalAssertionFailure() then
 6:       K.Add(ExtractLocationValue(c));
 7:    end if
 8: end for
 9: return K;
```

Fig. 5 Test case search worker algorithm

```
Input: Program C; Failing Test Case Queue F ≠ []
Output: Fault Location Set L
 1: (D, L) = ApplyGenericTransforms(C);
 2: VisibleCores = Min(Len(F), OS.VisibleCores);
 3: P = EstablishWorkerPool(VisibleCores);
 4: for 1 .. VisibleCores do
 5:    P.AddTask(D, Dequeue(F), L);
 6: end for
 7: WithoutImprovement = 0;
 8: while P.HasOpenWorkers() do
 9:    Sleep(TickTimerMS);
10:    if P.HasCompletedTasks() then
11:       Sleep(0.25 * P.GetFastestCompletedTaskTime());
12:       for w in P.GetCompletedTasks() do
13:          Lnew = w.Locations ∩ L;
14:          if Lnew == L then WithoutImprovement++;
15:          else WithoutImprovement = 0;
16:          end if
17:          L = Lnew;
18:       end for
19:       if WithoutImprovement > 15 then return L;
20:       end if
21:       F.Enqueue(P.ReturnTCsForIncompleteTasks());
22:       for w in P.GetAllTasks(); do
23:          P.RemoveTask(w);
24:          if Len(F) > 0 then P.AddTask(D, Dequeue(F), L);
25:          end if
26:       end for
27:    end if
28: end while
29: return L;
```

Fig. 6 Pool manager algorithm

ated assignment to `__t` and `sym()`. In the case of ESBMC, this process is iterative as only one counterexample is generated by each instantiation of the symbolic analyser. For ESBMC, a loop that modifies the input program to further narrow the toggle values explored allows the symbolic analyser to run repeatedly until no new toggle value is generated. This, executed inside the call to *CallModelCheckerOnInput* in Fig. 5, generates a full list of toggle values associated with repairs for this failing test case. KLEE does this loop automatically within a single call.

In lines 3 – 8 the old whitelist is replaced with the new list of toggle values returned by the counterexamples, generated by the symbolic analyser execution. This narrowing checks the counterexample type to ensure the assertion raised is the `assert(0);` added before the program's terminal nodes.

The time it takes to process this task is not predictable with any degree of certainty. The core operation of calling the solver inside the symbolic analyser is a logical satisfiability problem, which is NP-complete [16]. This type of SAT problem has been shown [13] to not provide predictable tractability. This unpredictability of the time it takes to process each logic expression in the symbolic analyser is the core issue that our algorithm must cope with. Each counterexample generated has a corresponding solver stage, and there are an unknown number of counterexamples multiplying this unknown per-instantiation processing time.

Each counterexample generated increases processing time and so a significantly narrowed whitelist, which blocks off many counterexamples, is highly beneficial. We manage uncertainty via the pool manager and ensure that narrowing results are percolated to new tasks as soon as possible.

3.3 The pool manager algorithm

The main tool loop, which manages the process pool and task scheduling, generates an output set of viable repair locations (lines 29 and 19 of Fig. 6). The input is an untransformed C program and a non-empty queue of failing test cases, each of which comprises an input string and a correct output string. To provide our evaluation of this tool with generalisation validity, the queue's order is randomised. This is because the optimised search varies in performance based on the test case order, as discussed in Sect. 4.2.

ApplyGenericTransforms in line 1 applies the generic transform stage discussed in Sect. 3.1. This parsing of the source file, as it walks all the fault locations being searched to apply the assignment transform, is also used to generate the initial whitelist of all toggle values that correlate to a localisation.

A worker pool is established in line 3, which provides the interaction point for all calls involving workers and the tasks they are executing or schedule for future execution. The tool queries the operating system to establish the multiprocessing pool is as wide as the exposed CPU core count, unless there are fewer failing test cases than available cores (line 2). The use of the worker pool is to avoid a single intractable task from stalling the entire search. This is most efficiently achieved on modern multi-core consumer hardware by dedicating one core to each worker. A similar pool could be managed on a single-core processor using the OS scheduler to manage the tasks, with the added overhead of regularly swapping the current process.

Each worker will process an independent task (i.e. Fig. 5) that takes the transformed program from line 1, a failing test case, and the current whitelist. This will eventually deplete the test case queue. Lines 4 – 6 push an initial batch of tasks to the pool system, which will use the un-narrowed whitelist created in line 1. Batching tasks with a multiprocessing implementation increases throughput, even with an algorithm dependent on the pruning of the search space for efficiency. Critically, this prevents one slow task, whose individual contribution is not required for the search, from completely stalling the full search. A typical four-core CPU executing highly variable return time tasks with probability p intractable outliers in the task queue will only stall the entire process when all cores are filled with intractable outliers at once. This probability of p^4 is a significant improvement, especially for this process where stalled tasks may become tractable later due to whitelist narrowing of the search space.

The main tool loop starts in line 8 of Fig. 6. The loop exits when the pool manager is not holding any completed tasks (waiting for their return value to be processed), no tasks are in flight, and no tasks are queued waiting to be started.

The pool manager thread sleeps to allow the pool's workers to monopolise the CPU, periodically waking to check whether any tasks have completed with *HasCompletedTasks*. When at least one completed task is ready for retiring from the pool, the main thread waits for other tasks to complete. This waits a maximum of 25% of the time the fastest task completed with (line 11). This allows slower tasks with valuable narrowing results to complete and be added to the narrowing before the next generation of tasks is dispatched. The completed tasks after this time-out are iterated in lines 12 – 18. To prepare for the next generation of tasks to be dispatched, a new whitelist is created that includes all narrowing returned from the completed tasks during this generation.

A counter, *WithoutImprovement*, is incremented if the returned narrowing does not prune the existing set. This will eventually trigger an early termination clause (line 19) when the narrowing process has stalled for many failing test cases in a row. This provides enhanced time performance with larger failing test case sets, without harming the narrowing performance, as the larger sets have more redundant (for narrowing) test cases. The choice of a threshold value can be tuned to the application and scale of the test suite involved, shown here with a suggested value of 15.

Any task that has been flagged as failing to complete in a time consistent with the others, missing the 125% dynamically assessed expected time, is queried in line 21. Any test case that failed to complete is added back to the tail of the queue. If the whitelist is smaller when this task comes back up, then it can be rescheduled, with the reduced search space increasing the likelihood of a fast completion time. Test cases are flagged as repeats so they are not enqueued a third time if they failed to complete a second time. This protects against intractable tasks that will not provide narrowing data. Remember that we only remove locations that cannot repair the fault, so failing to process a test case will only possibly lead to a worse localisation performance but we will never miss the true repair locations. All the tasks from a generation will have now been processed, so they are ejected from the pool (line 23). The next batch of tasks is despatched to the pool (line 24) with the newly narrowed whitelist, if there are still failing test cases in the queue.

This generational search process, with percolated combined narrowing, limits the explored search space to relevant branches where a find is still viable and reduces the symbolic execution work for the solver. Some tasks may still be intractable. To prevent them slipping through any cracks in this process, a global timer is established that ejects tasks that fail to complete in that time. This timer will only be triggered if all active workers are stalled with intractable problems, but it does require configuration of what is an unacceptable wait.

Scheduling this task pool over a standard consumer multicore CPU with these guards against search stalls and early rejection of superfluous searches provides significant performance improvements, as indicated by our results on the TCAS benchmark variants.

4 Fault localisation on Siemens TCAS benchmark

We initially demonstrate the time and localisation performance of our tool on the Siemens test suite's TCAS program and test universe taken from the Software-artefact Infrastructure Repository (SIR) [20]. For the single-fault variants, we maintain time performance within the same order of magnitude as the current model-based fault localisation state of the art, the algorithm by Jose and Majumdar [34]. We guarantee returning the location of the injected fault in every failing case, which Jose and Majumdar cannot. For 31 of the 33 single-fault variants, we improve on the localisation performance of Jose and Majumdar's results.

4.1 Data set

TCAS is a 173 line C program that provides a branching integer calculation that processes 13 input values into a three-option output decision. This is a loop-free program that contains 12 methods and has approximately 40 assignments, depending on whether implicit assignments are counted or not. The only C standard library calls used are to `atoi`, which is used to process the ASCII inputs into numerical values.

From the correct TCAS original, 41 variants have been generated by seeding (injecting) faults. Of these, 33 variants have been seeded with a single fault and exhibit at least one failing output with the accompanying test suite of 1608 test cases. These provide meaningful interpretation when comparing the performance of localisers with a single-fault assumption.

4.2 Experimental setup

Using TCAS, we compare five different approaches: the original Griesmayer et al. algorithm and our naïve reimplementation, our algorithm with two different back-ends (ESBMC and KLEE), and Jose and Majumdar's algorithm. In Tables 1 and 2 the headings refer to the following data sources and test platforms: Griesmayer's original data [23, §4, Table 1, p. 104] (**G**) uses CBMC on a 2.8GHz Pentium 4 and our naïve reimplementation of Griesmayer's algorithm (**N**) using ESBMC v1.17 on a 3GHz Core2Duo E8400. This reimplementation on a more modern Intel CPU and a recent SMT-based solver is designed to explore the potential performance improvements over an outdated version of CBMC and the Pentium 4. Our new algorithm using ESBMC (**E**) and KLEE (**K**) as back-end both ran on a 2011-era 3.1GHz Core i5-2400, with the ESBMC [17] v1.21 and KLEE [10,11] (for LLVM 3.4) symbolic analysers. Jose and Majumdar's results (**J**) are reconstructed from data provided [34, §6, Table 1, p. 443] using MSUnCORE on a 3.16GHz Core2Duo. Boldface entries in the table represent the best performance, underlined entries indicate failure to return the injected fault location for all failing test cases. In Table 1, the Jose and Majumdar time data has been calculated by taking the number of executions per test case and multiplying by the reported average time to complete a single execution of a failing test case.

We shuffle the test suite to randomise the order of the failing test cases when invoking our tool. This prevents the performance reported by our current tool only reflecting the time performance when provided with the default test case order. The time performance reported for our implementations (i.e. N, E, and K) are the averages of ten runs.

4.3 Run-time performance

Griesmayer et al. provided results on TCAS using state-of-the-art (for the time) model checking tools (CBMC) but indicated the design had not been optimised, saying "we do not concentrate on performance" [23, §4.1, p. 105]. We reimplemented this naïve process as described by Griesmayer et

Table 1 Seconds to return location set for test suite

	G	N	E	K	J		G	N	E	K	J		G	N	E	K	J
v1	2953	1442	9.0	4.5	**2.1**	v14	594	101	3.2	**1.4**	**1.4**	v26	311	114	3.4	2.1	**1.2**
v2	836	678	3.7	**3.2**	4.7	v16	1263	746	8.8	**3.9**	7.3	v27	153	107	3.3	2.3	**1.1**
v3	423	240	6.7	3.6	**2.2**	v17	1300	365	5.6	3.6	**3.4**	v28	642	711	**2.0**	2.7	6.1
v4	576	307	7.5	2.9	**2.7**	v18	499	188	3.7	**3.0**	3.6	v29	224	112	2.9	3.4	**1.7**
v5	159	106	3.2	2.3	**1.2**	v19	691	193	5.4	3.4	**2.1**	v30	939	508	3.9	**2.8**	3.7
v6	253	134	4.9	2.8	**1.3**	v20	748	196	7.4	4.3	**2.2**	v34	1906	790	4.9	**3.0**	7.7
v7	743	359	5.9	3.9	**2.6**	v21	585	197	6.7	3.7	**1.7**	v35	1069	711	**2.3**	2.8	4.6
v8	26	10	1.7	1.4	**0.1**	v22	223	42	2.8	2.6	**0.6**	v36	877	219	**2.1**	2.4	3.0
v9	114	72	2.0	2.1	**0.8**	v23	885	189	4.2	**2.8**	4.2	v37	822	729	**3.7**	4.1	**3.7**
v12	1664	727	5.0	**3.4**	11.5	v24	254	71	3.3	2.1	**0.6**	v39	66	8	1.0	1.5	**0.3**
v13	149	43	1.9	1.1	**0.3**	v25	68	8	1.0	1.5	**0.2**	v41	956	309	8.0	4.2	**2.4**

Griesmayer's original data [23, Table 1, p. 104] (**G**). Naïve reimplementation of Griesmayer's algorithm (**N**). New algorithm using ESBMC (**E**) and KLEE (**K**) as back-end. Jose and Majumdar's results [34, Table 1, p. 443] (**J**)

Table 2 Percentage of lines of code returned by localisation; visualised in Fig. 7; see Table 1 for legend

	G	N	K	J		G	N	K	J		G	N	K	J
v1	8.7	10.4	**7.5**	8.6	v14	**2.3**	2.9	2.9	8.1	v26	4.6	6.9	**4.0**	9.2
v2	**2.9**	**2.9**	**2.9**	4.6	v16	8.7	10.4	**7.5**	9.2	v27	4.0	8.1	**3.5**	10.9
v3	**4.0**	8.7	5.2	9.8	v17	2.3	**1.7**	2.3	9.2	v28	**1.2**	**1.2**	**1.2**	5.7
v4	8.7	11.0	**8.1**	9.2	v18	**2.3**	**2.3**	**2.3**	6.9	v29	**1.7**	2.3	2.3	5.7
v5	4.0	8.1	**3.5**	8.6	v19	2.3	**1.7**	2.3	9.2	v30	**2.3**	2.9	2.9	5.7
v6	7.5	11.0	**6.9**	8.6	v20	8.7	12.1	**8.1**	9.2	v34	4.0	5.8	**3.5**	8.6
v7	2.3	**1.7**	2.3	9.2	v21	8.7	12.7	**8.1**	8.6	v35	**1.2**	**1.2**	**1.2**	5.7
v8	11.0	16.8	10.4	**8.6**	v22	4.6	**4.0**	4.6	5.7	v36	**1.2**	**1.2**	1.7	2.9
v9	5.2	**4.6**	**4.6**	5.2	v23	5.2	**4.6**	**4.6**	6.3	v37	2.9	**1.7**	2.3	8.6
v12	**4.0**	4.6	**4.0**	9.2	v24	8.7	11.0	**7.5**	8.6	v39	4.6	**3.5**	4.0	6.9
v13	**5.2**	9.2	**5.2**	9.2	v25	4.6	**3.5**	4.0	6.9	v41	**8.7**	12.7	9.2	**8.6**

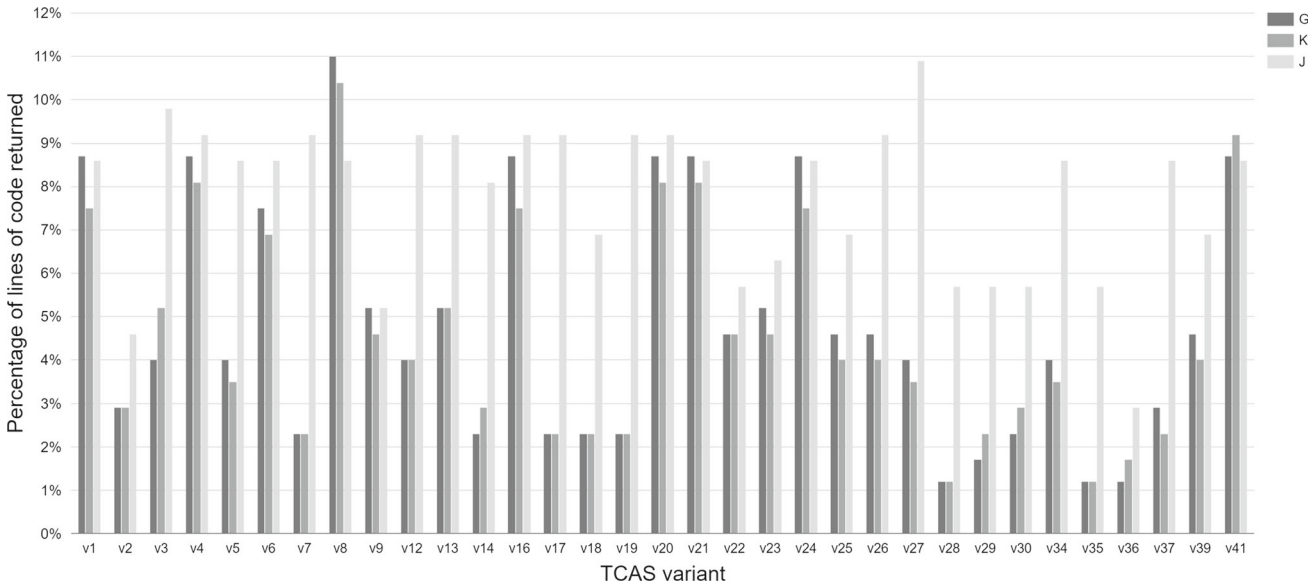


Fig. 7 Chart of data in Table 2; see Table 1 for legend

al.. This is running on more modern hardware and updated to use the current, CBMC-derived, SMT-based symbolic analyser ESBMC. We implemented automatic specification encoding into the tool to hard-code test cases. This tool iterates over all failing test cases, waiting for the symbolic analyser to return all flagged locations. The tool then returns the common locations flagged by all the failing test cases.

The average halving, and at most sixfold, decrease in completion time from Griesmayer's results (696 s average) to our naïve reimplementation (325 s average) in Table 1 shows some performance increase is derived from using a modern symbolic analyser on modern hardware. But, for example, variant 1 moving from over 49 min to over 24 min to return a location set is not viable compared to the 2.1 s of Jose and Majumdar or comparable with our optimised algorithm when also using ESBMC, at 9 s.

We have implemented our algorithm as tools interfacing with ESBMC or KLEE. As discussed in Sect. 3, this is designed to maximise consistency and avoid worst-case processing time, as well as reducing the symbolic execution burden to improve times. We provide run-time numbers for our algorithm using an ESBMC and KLEE back-end in Table 1. This indicates that using KLEE is often somewhat faster, compared to the ESBMC back-end, but the use of KLEE as the symbolic analyser is not a major factor in the orders of magnitude time performance gap between the naïve reimplementation of Griesmayer et al. and our algorithm with a KLEE back-end.

We maintain time performance within the same order of magnitude as the current model-based fault localisation state of the art, as presented by Jose and Majumdar, throughout the singe fault TCAS variants, marginally beating their times in ten of the 33 variants. Our tool, using the KLEE back-end, averages a completion time of 2.87 s per TCAS variant, compared to an average of 2.80 s in Jose and Majumdar's results. The ability of KLEE to scale to larger input programs offsets the few instances where it does not lead our tool's results using the ESBMC back-end.

These results support our claim that a Griesmayer-derived model-based localisation technique can be modified to be fast, comparable to the current alternatives when localising on some small integer programs such as the TCAS variants. Using intelligent pruning of the search space to minimise the symbolic execution load while minimising the disruption of a slow or intractable search node is facilitated by a multiprocess design that takes advantage of modern consumer processor architectures.

4.4 Localisation performance

The scope of the localisation of a tool quantifies which locations are being searched by the process and flagged as a potential fault. Different localisation scopes for each technique's implementation mean their localisation performance is not precisely comparable. The results published by Griesmayer et al. only explore the 34 explicit assignments in the TCAS variants, which increases the localisation performance we would expect to see in Table 2 as there cannot be more than 20% of the total lines of code returned. Our naïve reimplementation has an expanded scope that finds implicit assignments within the source code, expanding the potential locations returned to 43 assignments, or 25% of the source lines. This accounts for the weak, for Griesmayer-derived, localisation performance. The localisation results for our algorithm using the ESBMC back-end are omitted, but were noted to fall in line with the original Griesmayer et al. results and our current numbers with KLEE. Our current tool, using a KLEE back-end, does not apply all implicit assignment transforms implemented in our naïve reimplementation, only implementing the transforms described in Sect. 3.1. This reduces the assignments tracked to 39, or 23% of the source lines.

We can conceptualise the Griesmayer-derived searches as building a lookup table for each assignment location returned that, if complete, repairs all failing test cases. Passing test cases already have known correct values for their assignments. Any location flagged by a failing test cases will have, in the `sym()` value extractable from the counterexample, an assignment that repairs that trace and so converts the failing test case to a passing one. It is thus possible, for each location flagged by all failing test cases, to construct a complete lookup table that ensures every test case now has a specification-complying trace, i.e. a genuine repair exists at that location. In our results, the injected fault location is always included in the locations returned. But, with this conceptualisation of the process, the other locations are not false positives but additional locations where a genuine repair exists that will allow the test suite to pass, to the limits of the symbolic analyser's accuracy.

All our results confirm roughly comparable localisation performance between the various Griesmayer-derived methods, after accounting for the differences in localisation spaces. Any performance regression in localisation performance, when comparing the original results by Griesmayer et al. with our Griesmayer-derived localisation results, is most likely the result of searching a wider assignment space. Localisation performance improvements are likely to have resulted from more modern symbolic analysers providing a more accurate exploration of the input C program, exploring new potential traces. Exact localisation performance, while a common metric for comparison on TCAS and in general, is slightly defocussed as a primary metric here. Each location returned by a Griesmayer-derived tool is based upon a possible program repair that remedies the injected fault. Evaluating the difference between Griesmayer-derived techniques, as they all operate to generate this family of loca-

tions with lookup table justification, is to penalise a tool for returning justified fault candidates. Each such location has a genuine repair, despite not being the fault injection location.

Jose and Majumdar, with an approach based on mapping MAX-SAT clauses back to source code, cannot be directly compared in terms of potential C code coverage. The mapping of the MAX-SAT output, from logic clauses in the maximum satisfiable result to source locations, can flag locations other than assignments. However, the granularity of this mapping is not clear. Some of the lack of competitive localisation performance in some variants shown in Table 2 for Jose and Majumdar, when compared to Griesmayer et al. or our algorithm can be explained by this different scope of potentially returned locations, where additional potential repair locations are being suggested outside of assignment modifications.

Comparing the localisation performances, even without being "apples to apples", is ultimately comparing sets of proposed fault sites where human developers must search for a genuine repair. Our current tool is typically ahead in this metric, sometimes by a significant percentage. In the two variants where our tool performs worse, *v41* returns a set of locations only one larger than those returned by Jose and Majumdar, and *v8* returns a set three locations larger.

When comparing the localisation performance of these tools, we must consider that there is an injected fault location for each of the single-fault variants of TCAS. For all the Griesmayer-derived techniques, then the injected fault location (the location where a variant was seeded with a fault) is always included in the returned list of locations. Due to the technique's design (where a location is only returned if it is common between all individual failing test cases), this means that this injected location will also be returned when only given a single failing test case from any of the test suites and on any of the single-fault variants. Any subset of the test suite that contains at least one failing test case will, for all Griesmayer-derived tools, flag the injected fault location.

The results from Jose and Majumdar cannot make a similar claim. To account for some failing test cases not indicating the injected fault location, their final set is based on the most commonly indicated locations, not locations that are always flagged by every failing test case. Their results for each full test suite do flag the injected fault location for TCAS as most common. But they indicate that there exist subsets of the failing test cases for which they would not flag the injected fault in the case of six single-fault variants (underlined in the tables).

4.5 Comparison to spectrum-based fault localisation

Spectrum-based fault localisation techniques, compared in [45,69,71], are dynamic analysis methods that operate by examining the traces of passing and failing test cases. They assume that faults are more likely to be exercised by failing test cases and less likely to be exercised by passing test cases. The statements in a program can then be ranked based on different weighting techniques.

Table 3 TCAS average rank percentage as reported in [45, Table XI, p. 11:23] with corresponding name and maximal group used in [71]

Xie et al. [71]	Maximal?	Naish et al. [45]	TCAS
Naish1	ER1	O	9.90
Naish2	ER1	Op	9.90
	No	Tarantula	10.80
Russel & Rao	ER5	Russell etc.	14.47
Binary	ER5	Binary	14.47

Probably the best-known formula is the Tarantula formula [33]. While it has been shown to not be maximal [72] the difference between a maximal approach and Tarantula in TCAS is 10.8% average ranking versus 9.9% [45, Table XI, p. 11:23]. In contrast, model-based localisations rank the injected fault at an average of 2.3% (our tool) to 3.9% (Jose and Majumdar) when looking at the single fault subset of variants. In Table 3, we show the results from Naish et al., which have been cross-referenced to the formulas later found to be maximal by Xie et al.. These results show the average percentage ranking based on percentage of all lines of source code, with 0% being the best result and 100% being the worst. We have also included the non-maximal Tarantula formula in this table to provide context for the relative performances of these formulas on the TCAS benchmark variants discussed here.

Our approach does not rank locations. In order to compare our KLEE results from Table 2, they must be converted using a middle-line strategy, ranking equally suspicious statements with the mid-point rank. For example, we rank statements with suspiciousness (0.1, 0.4, 0.8, 0.4, 0.4) as (5th, 3rd, 1st, 3rd, 3rd). Results from our model-based tool are converted from an unranked list using this strategy, giving a suspiciousness of 1.0 to all locations returned and a suspiciousness of 0.0 to all others. An alternative way to conceptualise this result from the middle-line strategy is to consider an average of all random shufflings of a list. Any unranked list, when assigned a random order, will rank any item on the list at an average position half way down the list.

Comparing our results directly with those published previously that use spectrum-based localisation techniques is even more imprecise than the issues of narrowing comparability noted in Sect. 4.4 above. The granularity of most spectrum-based localisation techniques is down to the line of code as this is how code coverage tools like GCov [22] report their statement coverage output. All locations are ranked by suspiciousness using the various weighting formulas.

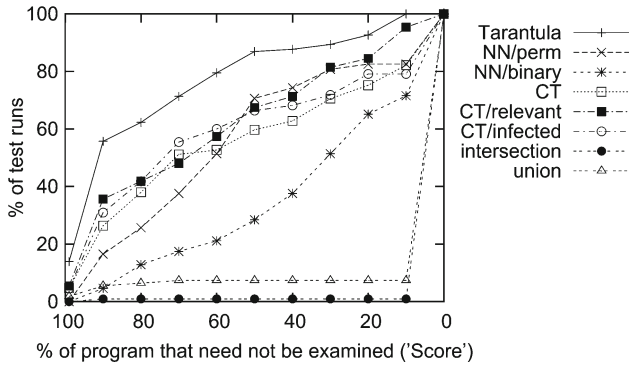


Fig. 8 Comparison of effectiveness of spectrum-based techniques reproduced from Jones and Harrold [33, Figure 2, §3.3]

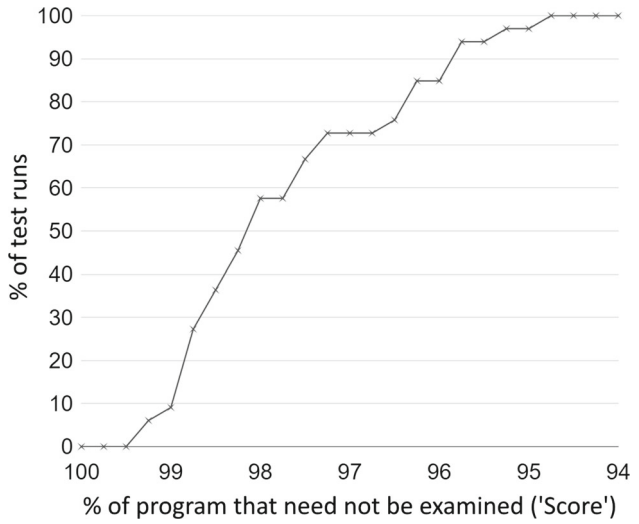


Fig. 9 Effectiveness of our localisation techniques (partial x-axis)

The localisation performance is measured by Jones and Harrold as the average percentage of the program that *does not* need to be traversed by the human debugger before encountering the inserted fault. A score of 100% then means that the inserted fault in the variant is marked as the most suspicious location, so a human debugger does not need to view any of the non-faulty code lines before encountering the injected fault. In Fig. 8 the term "test run" refers to a "program variant with injected fault". We have retained this labelling when providing our results in Fig. 9 for consistency. The y-axis can also be read as "percentage of variants localised" when examining the ranked list of locations to the depth noted on the x-axis. For example, the third data point for Tarantula in Fig. 8 indicates that for 61% of program variants, Tarantula ranks the injected fault above at least 80% of other statements and so a human debugger would find that injected fault without examining 80% of the program statements. The choice of label name, axis order, and score method is replicated here to retain consistency with earlier papers that establish this convention [14,33].

Jones and Harrold provide results based on the full Siemens suite of programs which is visualised as a chart of the percentage of the program that need not to be examined, i.e. the percentage of the program that is ranked as less suspicious than the injected fault location, against the variants that meet this performance criteria. There are a total of 132 different program variants over the full Siemens suite being summed here so the performance of only the 41 TCAS variants are a minority of the results reported. For this reason, we have not overlaid our results onto the reproduced chart, as the 'test runs' (i.e. program variants) are not equivalent. However, in later empirical results [45, Table XI, p. 11:23], the TCAS variants are reported to average the rank of 88.85% with Tarantula, the second worst average ranking of all the Siemens Test Suite programs reported in that paper. This indicates that the TCAS results are not unfairly represented by being collected with other Siemens results in Fig. 8. It should also be noted that each of the lines on the reproduced chart are also referring to slightly different subsets of the total 132 program variants in the Siemens suite as those that cannot generate a result are rejected. Tarantula reports on 122 variants [33, §3.2.1], Cause-Transitions (CT) on 129 variants [14, §7.5], and Nearest Neighbour (NN) on 109 variants [33, §3.2.1].

The chart is constructed into decile buckets except for the initial 100% bucket as not traversing any of the program under test will guarantee that the fault has not yet been traversed. This is replaced by a 99% bucket for variants that have been localised with the injected fault in the top 1% of all locations and a bucket below it for those over 1% but under 10% (98–90% in the scale used). The comparison shows how Tarantula outperforms some contemporary techniques and so this solid line is the one we compare our results against. We have reproduced this in Fig. 8. It is clear that the narrowing, while excellent for a few and good for many when using Tarantula on the Siemens variants, does not reach a point of uniform guarantee before the majority of code must be searched. Some variants are localised with a result that ranks the true fault location as less suspicious than most of the rest of the locations in the program, a result worse than chance.

However, charting our results using the same decile buckets provides an uninteresting comparison. This is because, by the 90% plot, our tool has ranked every variant's injected fault inside of the list. An average traversal of a ranked list generated with our unranked conversion will have passed the injected fault before 10% of the lines have been looked at for every variant. Even looking at the absolute worst-case performance of our tool on the single-fault TCAS variants, assuming our random ranking of the list of returned locations always put the injected fault location as least suspicious, we would still have reached 97% of the variants covered at the 90% need-not-be-examined point on the chart.

We have therefore provided in Fig. 9 a zoomed-in plot of only the leftmost 6% of the full chart range and have bucketed this data at every quarter percentage. This indicates how our model-based localisation results would be represented on the very leftmost sliver of Fig. 8 before running a straight line at 100% of the variants for the majority of the chart. This should not be read as a definitive comparison and is provided with significant caveats. This data is not directly comparable due to variable code coverage criteria and because the chart from Jones and Harrold combines data from various large subsets of all the Siemens variants, not just the 33 single-fault TCAS variants we are charting. However, it still shows that our method specifically outperforms spectrum-based methods.

4.6 Threats to validity

In Sect. 4.4 we have already discussed issues with making direct localisation performance comparisons. These are not direct "apples to apples" comparisons as each localisation method brings with it various limitations and assumptions. The granularity and fault classes being searched for vary between the different model-based and spectrum-based localisation methods compared in this article. Different localisation scopes for each technique's implementation mean their localisation performance is not precisely comparable which is a caveat noted whenever we have presented a comparison.

The single-fault assumption that underpins our model prevents any meaningful localisation performance on the seven TCAS variants that contain multiple injected faults, where positive localisation results would be derived from blind chance. The performance of a single-fault localiser will be faster than more extensive searches that include k-fault analysis. However, the single-fault assumption is common in fault localisation techniques [12,23,34].

The fault-seeded variants of the small Siemens program we are testing on are not a representative sample of C programs and the faults they contain. The TCAS variants explored all contain injected faults inserted at return expressions or assignments. Our results may not generalise to other C programs. Our focus on a subset of programs, and use of real-world code that is atypical in the heavy use of global variables, may obscure comparative analysis of performance against other tools with different program features. Performance on relatively small, loop-free programs like TCAS does not provide guidance into how this process scales to large programs with more complex control flow. However, this issue is common to all tools that demonstrate their localisation effectiveness on the TCAS variants.

We can use any (C99 comprehending, supporting `assume` functionality) symbolic analyser to process our generated C code but our results are linked to either KLEE or ESBMC. Any issues related to those tools may affect our results, if not our methods/process. No high performance symbolic analyser of C can perfectly transform an input program into an exact representation according to the full C specifications. The lack of exact specification compliance by the various widely used (optimising) C compilers also makes such an impracticable achievement undesirable. Real compiled code does not perfectly map to a strict adherence to a single, deterministic interpretation of the C specifications.

To minimise the risk of over-tuning our design to the test data, our choice of time-out, sleep delay, and early termination values have not been tuned or selected for optimising with respect to the data set as this would compromise the generalisability of our results.

5 Fault localisation on student programs

In an educational setting, introductory programming courses typically require instructors to assess a large number of small programs by novice programmers. Many of these programs contain errors, ranging from small mistakes to complete design and implementation failures, reflecting the students' misunderstanding of the task or their solution attempt. With limited resources, greatly enhanced learning outcomes are achieved if students are (in the former case) automatically directed towards the locations of their mistakes to allow self-guided repairs, so that the instructors can focus (in the latter case) on addressing fundamental misunderstandings. However, existing basic assessment tools (such as Ceilidh [4], ASSYST [31], and BOSS [35]) based on simple compilation tests, test suites, or model solutions do not provide the detailed feedback necessary for self-guided repairs and do not support instructors in quickly separating solutions with small mistakes from complete failures. Compilation tests do not give any feedback for syntactically correct programs, test suites can give misleading results if programs contain simple errors that affect a large number of test cases, and model solutions cannot account for the variability of student programs. In this section we apply our fault localisation approach to a corpus of student programs and show that its performance, both in terms of run-time and localisation precision, would allow its use in practice. Specifically, our approach can provide feedback to enable self-guided repair and can locate programs that "almost" conform to question requirements but are scored badly by functional grading using test suites.

5.1 Data set

We use a data set of around 30k passing and 150k failing Java and Python programs collected by the automated assessment system of the University of Auckland [53]. These programs were written by students to answer 1693 different Computer Science coursework questions. We translated

the programs into a C representation using a custom-made simple converter that was designed to retain the style and functionality of programs without translating any detailed differences between the languages, such as language-specific handling of integer overflows. This translation would allow the mapping back of locations to the original Java or Python source code, although our model-based downstream components currently only reason within the specifications of the C programming language they have been translated into. We consider the translated programs in C to be the corpus on which we are testing the efficacy of our tool. Our methods are more directly applicable for native reasoning on Java or Python programs by using suitable downstream components designed for those languages.

We rejected programs that produced translation failures (typically due to unsupported standard library calls), that use floating-point arithmetic (which is not yet supported for symbolic exploration by our downstream components), that comprise less than three assignments (to avoid trivial localisation tasks), or that KLEE or GCC's GCov (which we use as downstream components) could not process (such as programs containing infinite loops). This gave a set of 7000 failing student submissions to questions that the student had later also submitted a passing program. We then analysed the data set to identify pairs where a student had submitted a failing program that, after a single assignment edit, was later found in the database passing the full test suite. Such programs are guaranteed to contain a single fault (with respect to the full test suite as specification)—the changed assignment and thus allow us to apply the methods described in the previous sections without any changes (Sect. 5.6 shows an extension to dual-fault programs). This yielded 304 pairs that were answers to a range of different coursework questions. These programs contain an average of 5 assignments in 11 statements (as counted by GCov).

We ran a script to analyse each provided test suite (averaging 7.8 tests per pair) and generated new test suites that randomly picked inputs within an order of magnitude of the existing values. The passing program from the code pair was used as oracle to establish desired output values. These large test suites average 154 tests per pair, with 80 of those tests failing.

5.2 Experimental setup

We used the publicly available Hawk-Eye tool [27] to compare our model-based fault localisation to spectrum-based methods. Hawk-Eye implements the Ochiai and Tarantula formulas [71], which have both been shown to not be maximal [72]. We therefore extended it to implement the maximal Russel and Rao, Naish 2, and Wong 1 methods, as shown in Table 3. However, as in the case of TCAS, there are only minor differences between the different formulas over our student data set, due to the relatively simple structure and small size of the programs. Often, student code submissions execute every line of code for every test case, preventing a spectrum-based approach from discriminating between locations. More specifically, while the different formulas generate different suspiciousness values for the individual lines, they all return the same actual ranking for the injected faults line in any of our tests. Therefore we refer to the results collectively simply as the Hawk-Eye results.

Results from our model-based tool have been converted from an unranked list using the same middle-line strategy (where all locations returned are given a suspiciousness of 1.0 and all others a suspiciousness of 0.0). Absolute localisation performance is reported as the average rank of the fault location in the ranked list.

Percentage localisation performance is reported as the average percentage of other locations that will be searched before the fault is found when iterating over the locations in rank order. Hence, a score of 100%, the worst possible score, indicates that every returned statement would be searched before the fault was reached. A score near 0%, indicating very few locations ranked above the location used by the student to repair the submission in the database, means fewer instances of the debugging process stalling before repair synthesis can begin. Note that this is in contrast to the earlier results reported in Sect. 4.5. Note further that in Sect. 4.4 the *percentage of lines of code returned by localisation* is the total percentage of unranked lines (not searched locations) returned as potential repair sites.

We generated all results on a 3.1GHz Core i5-2400 using the KLEE 1.1 symbolic analyser.

5.3 Results for original test suites

On the 304 selected (i.e. failing) student submissions and using the instructor-authored test suites, our tool returns the faulty assignment after, on average, only 16% of other assignments when providing a ranked list of locations. In Fig. 10 these rankings are divided into two per cent buckets to show the distribution. A lower percentage score means less of the program must be manually explored by a programmer before the faulty assignment is encountered. Our tool regularly pinpoints the assignment later used by the student to bring the submission into compliance with the test suite. 125 of the 304 student submissions were returned from our tool with only that assignment flagged (leftmost bucket in the histogram with a 0% percentage localisation), the ideal result [48]. In these cases, all other assignments were eliminated as viable repair candidates for the test suite specification.

The chart in Fig. 11 collects the program's assignment count (how many locations could be ranked by our tool) against the absolute localisation rank (the position in the ranked list of the fault later repaired by the student, using a

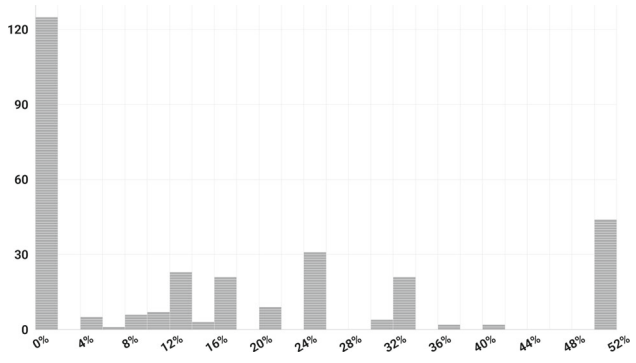


Fig. 10 Histogram grouping programs by their percentage localisation performance (relative rank of injected fault, x-axis) using the original test suite

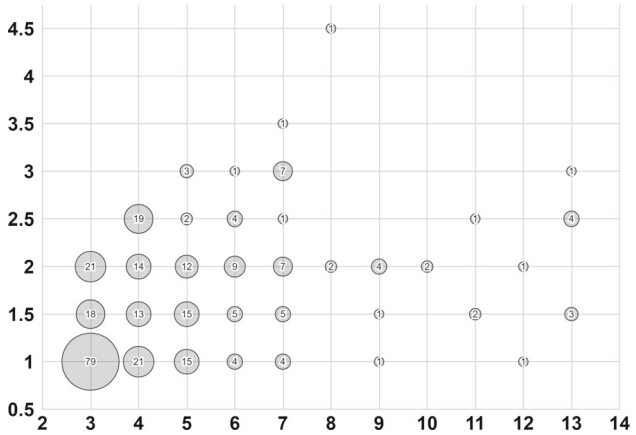


Fig. 11 Bubble plot counting programs by absolute localisation using the original test suite (x-axis) versus assignment count

mid-line strategy when several locations are ranked equally). For example, 15 programs with five assignments ranked the repaired assignment equally first in suspiciousness with another assignment, resulting in an average rank of 1.5 for the assignment later used to repair the program. This demonstrates that small programs (assignment counts of 3 to 5), which make up the majority of the student submissions tested, do not skew the performance of our tool towards overestimating the absolute narrowing ability of our technique. In fact, looking at the larger assignment counts (8 and above) indicates that percentage localisation performance is weakest for low assignment count programs. These smaller programs, where consistently strong absolute localisation performance is limited by the small list of potential locations, lead to comparatively weak percentage localisation performance. Since the absolute number of locations shown before the actual repair site is critical to facilitating debugging progress [48], strong absolute localisation performance throughout the data set is highly desirable. This topic is discussed further in Sect. 6.2.

Hawk-Eye ranks the statement the student later used to repair the program after an average of 63% of other statements. This is 47 percentage points adrift of our tool. No correlation was found between the program size and the comparative performance of our tool compared with Hawk-Eye.

On average, Hawk-Eye is provided eight test cases (passing and failing) for each submission, which are used to rank eleven statements. Since our tool only requires failing test cases, it is provided with an average of four failing test cases and localises over the average of five assignments in each submission. These localisations are produced in an average of 0.3 s per submission by both our tool and Hawk-Eye. Twice our tool hit too many pathological cases to adapt and only provided results at the tool's 10 s time-out; these results were masked in the average times by an otherwise slightly faster return time.

The limited number of failing test cases did not hinder our tool on this sample of short, real-world student programs. This strongly supports the use of our tool for assisting students in repairing single-assignment faults in small program submissions, even when only specified by a very small test suite. When a student has made a mistake on constructing an assignment in an otherwise solid submission, an expensive debugging process may be averted by use of this feedback with consistently high absolute localisation performance over the range of student program submissions.

On this selection of actual student submissions that were later brought into full compliance with the test suite with a single assignment edit, our tool provides strong localisation performance, either looking at absolute rank or percentage rank results. We also provide equivalent run-time performance on these localisation tasks when compared to a spectrum-based tool, while ranking the location later used to repair the program significantly higher.

5.4 Results for extended test suites

The provided test suites for such student courseworks were hand-written to provide high coverage with very few tests. The original weakness our tool is designed to overcome is only exposed when localising using a large number of test cases, where traditionally spectrum-based techniques have significantly outperformed a Griesmayer-derived localisation. We therefore significantly expanded the test suite, as described in Sect. 5.1.

Our tool, provided with an average of 80 failing test cases, did not improve from the 16% localisation score achieved previously. Both the distribution of percentage localisations plotted in Fig. 12 and the plot of the assignment count (x-axis) against the absolute localisation rank (y-axis) in Fig. 13 shows that the localisation performance does not significantly shift when using the extended test suites. Running the average of 154 test cases through each submission, Hawk-Eye

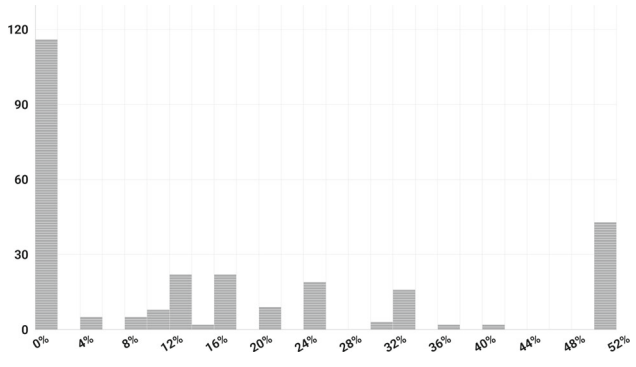


Fig. 12 Histogram grouping programs by their percentage localisation performance (relative rank of injected fault, x-axis) using the extended test suite

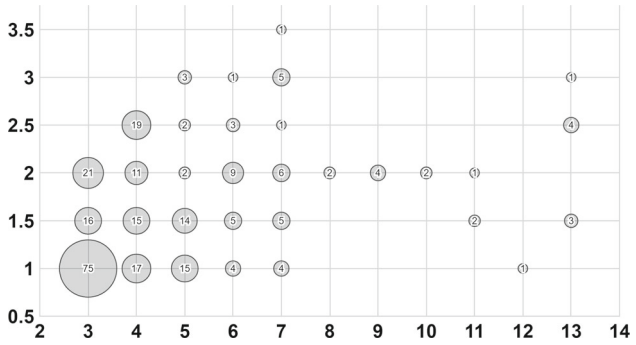


Fig. 13 Bubble plot counting programs by absolute localisation using the extended test suite (x-axis) versus assignment count

showed a modest improvement, although it continued to lag our tool significantly, at a 42 percentage point deficit.

The consistent localisation performance of our tool is due to a combination of different factors. Over a third of these submissions, with the smaller test suite, already provided perfect results using our tool; some programs will contain several assignments where a genuine repair is possible so there is no more compact list of repair locations; and some programs are resistant to analysis by symbolic analysis, which does not change with test suite size. This may have provided very little room for improvement by our tool.

However, the run-time on these much larger test suites confirm the scalability of our tool, an area where model-based fault localisation has traditionally suffered [23]. Our tool averaged 0.9 s per submission while Hawk-Eye lagged behind, averaging 1.1 s. Included in that average, three times our tool hit too many pathological cases to adapt and only completed at the time-out.

When significantly extending the test suite size from the instructor-authored suites, our tool retains the same strong localisation performance, looking at absolute rank or percentage rank results. While the spectrum-based technique does improve with the addition of many extra test cases, those results still rank the location later used to repair the program significantly below the ranking provided by our tool. We also provide faster run-time performance on these localisation tasks when compared to a spectrum-based tool, taking the lead due to our efficient scaling thanks to our optimised search process.

5.5 Syntactically richer programs

The simple converter discussed in Sect. 5.1 translated many student programs to an equivalent C representation. However, this translation was only designed to provide simple conversion and not to convert concepts that varied between programming languages, in order to guarantee that the resulting C program has the same behaviour as the original Python program, and in particular, that the nature of the test cases (i.e. passing or failing) does not change. Python features such as the use of tuples for return values and list comprehension are often used in simple programs but do not have beginner-level equivalents in C. We took several larger (20+ assignments) integer programs from the database that had failed the translation phase and manually translated them, retaining the spirit of the code while using C language features. These translated student submissions, see for example Fig. 14, were integer programs that contained `for` loops and C standard library calls, exercising new feature of the underlying symbolic analyser, KLEE.

The behaviour inherited from the underlying tools in regard to loop unwinding and recursive function calls operates on the transformed program which includes our symbolic insertion. The design of that transformed statement means that if a location is triggered by the toggle variable, all instances are switched to use symbolic values to find a potential repair. When a loop is unrolled, each instance of the assignment will be activated by the same toggle value test. Each activated assignment evaluation will generate a call to generate a new symbolic value, thus not restricting the range of repairs that this model explores. Recursive and multiple calls to an assignment-containing function are inlined by the symbolic analyser and so work similarly, allowing repairs to be found where the assignment needs to be changed each time it is called from the original program behaviour.

In the manual language translation process, naming conventions were used to direct the model transformation stage to ignore loop iterators generated when translating array and tuple assignments. These C loops exist as a single assignment in the original program, so no equivalent iterator exists to map back to, i.e. it is not a site of a potential repair. We also provided a contraction for these simple loops that only assigned to an array. In line 40 of Fig. 14, the toggle test has been extracted from the loop in line 41, removing the duplication of testing this condition as it was originally expressed as a ternary operator on the array assignment in line 41. This

```
1  static unsigned int __t;
2  static int coins[5];
3  void method(int change, int *__result) {
4    int twoDollar = (__t==0)? sym() : change/200;
5    if (twoDollar >= coins[4]) {
6      change = (__t==1)? sym() : 200*coins[4];
7      twoDollar = (__t==2)? sym() : coins[4];
8    } else
9      change = (__t==3) ? sym() : 200*twoDollar;
10
11   int oneDollar = (__t==4)? sym() : change/100;
12   if (oneDollar >= coins[3]) {
13     change = (__t==5)? sym() : 100*coins[3];
14     oneDollar = (__t==6)? sym() : coins[3];
15   } else
16     change = (__t==7)? sym() : 100*oneDollar;
17
18   int fiftyCent = (__t==8)? sym() : change/50;
19   if (fiftyCent >= coins[2]) {
20     change = (__t==9)? sym() : 50*coins[2];
21     fiftyCent = (__t==10)? sym() : coins[2];
22   } else
23     change = (__t==11)? sym() : 50*fiftyCent;
24
25   int twentyCent = (__t==12)? sym() : change/20;
26   if (twentyCent >= coins[1]) {
27     change = (__t==13)? sym() : 20*coins[1];
28     twentyCent = (__t==14)? sym() : coins[1];
29   } else
30     change = (__t==15)? sym() : 20*twentyCent;
31
32   int tenCent = (__t==16)? sym() : change/10;
33   if (tenCent >= coins[0]) {
34     change = (__t==17)? sym() : 10*coins[0];
35     tenCent = (__t==18)? sym() : coins[0];
36   } else
37     change = (__t==19)? sym() : 10*tenCent;
38
39   int result[5] = {twoDollar, oneDollar,
         fiftyCent, twentyCent, tenCent};
40   if (__t==20)
41     for (int i=0; i<5; ++i) {result[i] = sym();}
42   memcpy(__result, result, 5*sizeof(int));
43   return;
44 }
45
46 int main(int argc, char **argv) {
47   __t = sym();
48   assume(__t < 21);
49   int __out[5];
50   for (int i=0; i<5; ++i) {
51     coins[i] = atoi(argv[i+2]);
52     __out[i] = atoi(argv[i+7]);
53   }
54   int __result[5] = {-1};
55   (void)method(atoi(argv[1]), __result);
56   for (int i=0; i<5; ++i)
57     {assume(__out[i] == __result[i]);}
58   assert(0);
59   return 0;
60 }
```

Fig. 14 Example syntactically richer program

is consistent with the spirit of the Python program where this is a single assignment line that maps to the deterministic assignment in line 39, the toggle test in line 40, and the symbolic assignment in line 41 if that toggle is activated.

We selected five representative student submissions, single-fault programs with 20, 21, 21, 27, and 30 assignments each. Two of the programs had their assignment to a 5-wide array contracted to a single toggle as detailed above, retaining the spirit of the single assignment step in the original language. They are 44, 50, 55, 56, and 57 lines of C code long, measured by GCov. They answer two questions in the database; one of which has a test suite of 9, the other has 10 test cases. Of these, between 6 and 10 fail for each submission.

Of the five programs tested using our tool, three narrowed to only flag the single assignment later used by the student to bring the program into full compliance with the test suite (a 0% perfect ranked narrowing), one failed to provide any narrowing (equivalent to 50% rank narrowing), and one timed-out without returning any information. The completion time for the four programs that did not time-out were 0.4s, 1.7s, 1.8s, and 2.1s. When using Hawk-Eye, all five programs provided consistent return times but did not rank the faulty assignment as highly suspicious. This ranked the repair statement after an average of 32, 33, 35, 42 and 47% of other statements. Each result was returned in approximately 0.4 s.

This suggests that where our model-based method can gain traction, it can reason to provide precise results that pinpoint viable repair locations. For three of the five larger looping programs, our tool provided only the location later used to repair the submission while working with fewer than ten test cases as specification. But some programs transform into a model that is intractable for current solvers or does not provide valuable narrowing information. The program that fails to provide any results locates the majority of the program inside a `for` loop and also modifies the iterator variable in an assignment in the body of the loop. This provides significant issues for a state space exploration when the iterator is assigned a symbolic value as part of the transformation process.

5.6 Dual-fault localisation

The currently discussed algorithm uses a single toggle value that activates a single location, performing the alternative assignment (of a symbolic value). Our whitelist prunes locations based on this single-fault assumption, where a single test case not being repairable causes the location to cease to be searched. If multiple toggles were used with

```
(__t1 == 3 || __t2 == 3)
```

conditions activating the modified locations, then the search would be able to produce localisations searching for multiple faults (up to the number of toggles inserted and chained in the or-conditions). This multiple toggle method was proposed as an extension by Griesmayer et al. [23] to provide results for n-fault programs. This increases the solver cost due to additional non-determinism and would also increase the total combination of counterexamples returned, which may severely limit the applicability of this technique to very small input programs in order to retain tractability. We therefore explored an extension to our narrowing whitelist that would

```
1  int main(void) {
2    /* ... */
3    int z=14*a*c;
4    if((bb-z)<0) {
5      return pack(co1, co2);
6    } else {
7      return pack(re1, re2);
8    }
9  }
```

Fig. 15 Potential dual-fault program example

allow the collection of more results without using multiple toggle values to explore an n-fault assumption.

To explore student submissions pruned from our original slice due to containing more than one fault, as explained in Sect. 5.1, we reprocessed the entire database with slightly tweaked parameters. In this new slice, we rejected submissions using the same criteria except identifying pairs where a student had submitted a failing program that, after exactly *two assignment edits*, was later found in the database passing the full test suite. This yielded 125 pairs. These programs contain an average of 4 assignments in 12 statements (as counted by GCov), making them of the same scale as our initial slice of single-fault programs.

We analysed these pairs by performing a manual code review. The majority of these pairs, 77, contain two faults that cannot both be executed in the same trace. We characterise this class of dual-fault programs as *twin-fault* programs, which have a test suite that mixes the two fault-exercising subsets of test cases. The remaining 48 pairs are defective programs where traces from failing test cases will either always or sometimes execute both fault locations during a single trace. We call this class *double-fault* programs. In the example in Fig. 15, if lines 3 and 5 are incorrect then this would be a double-fault program where some test cases would exercise both lines and others would only exercise faulty line 3 (and then exiting via line 7). However, faults in lines 5 and 7 would make it a twin-fault program as all test cases would either only exercise line 5 or only line 7, no trace can exercise both faults.

For this new data set, the original test suites contain an average of 8.8 test cases per program, 5.7 of them failing. This can be insufficient to exercise each pair of assignments in a dual-fault program and classify the pair as twin-fault or double-fault. In Fig. 15, if the omitted code (shown as line 2) includes a branch that provides three paths, then simply covering each different path through this small program fragment requires six failing test cases, assuming no duplication of trace paths. Although program flow analysis can detect when a pair of toggled locations being searched will result in a twin-fault program, this analysis can be expensive. In fact, classifying the test cases for each assignment pair with a concrete approach (i.e. executing the program instrumented by GCov and collection which assignments are activated by each test case) will mean running a code coverage tool, as used in spectrum-based fault localisation, with similar time costs. We generated extended test suites in the same manner as with the original data set, discussed in Sect. 5.1. These contain an average of 270 test cases per program, 94 of them failing.

To adapt our tool to search for localisation results in twin-fault and double-fault programs, the whitelisting discussed in Sect. 3 must be replaced. A failing test case that exercises line 5 of Fig. 15 cannot flag line 7 as a potential repair location (as it is not exercised at all) so the lack of a localisation does not remove it from the search space for future test cases. Dual-fault programs require the consideration of previously completed localisations in order to reason about the possible narrowing of the whitelist of locations. If previous failing test case have always either flagged line 5 or 7 but not other lines then, under a twin-fault assumption, we can eliminate other lines from the search space.

However, this whitelist adaptation can, for some programs, lead to no narrowing of the whitelist for the entire duration of the localisation run as no location can be ruled impossible to be one of the two repair sites. This assumption also cannot localise with a failing test case that exercises both locations on double-fault programs; this results in no locations being flagged as a possible repair site. If lines 3 and 5 must always both be changed to fix any failing test case in Fig. 15, then no single toggle value can exercise that widened symbolic behaviour to find that repair. Those negative localisation results must be ignored (dropped) rather than integrated into the narrowing process.

To accelerate the process for larger test suites, we rank the frequency in which the locations are capable of repairing previous failing test cases and use early narrowing to reject locations that have been flagged significantly below the bulk of other locations. This optimisation allows early search space reduction with a very low chance of erroneous rejection in much the same way the early termination in Sect. 3 does. This ranking is used to provide a ranked list of all locations with associated suspiciousness, allowing our tool to provide ranked results (cf. Sect. 5.2 where our original algorithm only provides one and zero suspiciousness levels to any assignment). These changes produce a search that narrows less often and less early.

5.7 Results for dual-fault localisation

In Table 4, after submitting the dual-fault data set to our adapted tool, we analysed (via manual code inspection) and split the results into twin and double-fault subsets. This split was based on our knowledge of the repair locations used by students to later bring the programs into full compliance with the test suite. We present the results using the original test suites, where the early narrowing optimisation does

Table 4 Dual-fault localisation performance

	2P	2nP	1P	1nP	F	Loc (%)	Rank	Time
Twin (orig)	47	24	4	2	0	24	2.0	0.3s
Twin (ext)	37	32	4	4	0	24	2.0	2.5s
Twin (ext, opt)	40	31	4	2	0	24	2.0	1.0s
Double (orig)	7	34	3	4	0	34	2.3	0.8s
Double (ext)	6	35	3	4	0	36	2.3	4.2s
Double (ext, opt)	6	35	3	4	0	37	2.3	2.0s

2Perfect double localisation; non-perfect double (**2nP**) localisation; **1P**erfect single, localisation; non-perfect single (**1nP**) localisation; **F**ailed localisation; mean **Loc**alisation rank; mean absolute localisation **Rank**; completion **Time**

not have time to trigger, and for extended test suites both with and without the (early narrowing) ranking optimisation discussed above. We have calculated averages of: the mean absolute localisation rank (**Rank**) of the two faults later used to repair the program; the completion time (**Time**) to run localisation on a program with all failing test cases; and the mean localisation rank (**Loc**) of the two faults as a percentage of the full range of rankings possible.

We also divide the 77 twin-fault programs and 48 double-fault programs by the final whitelist results they generated. This classifies our results based on our hybrid approach, described above, where locations are both given suspiciousness ranks and narrowing reduces the list of locations returned. Perfect (**2P**) results flag exactly two locations, both of the faults used to later repair the program to bring it into full conformance with the test suite. Non-perfect (**2nP**) results flag both locations but include other locations as possible repair sites, which is often due to other genuine repairs existing. Some of these non-perfect results rank the two fault locations most highly (i.e. *Loc* of 0%, *Rank* of 1.5) despite also marking other locations as viable repair sites.

Suppose the program shown in Fig. 15, with a twin-fault in lines 5 and 7, was localised with a test suite that only exercised line 7. A result where the suspiciousness of all locations other than line 7 was zero while line 7 was 100% is plausible. In this case, the dual-fault search indicates a single-fault repair solution has been found as the only repair location. There also exists dependent assignment fault chains, for example `a = x*y; b = a/z-p;`, which generate single-fault repairs (at `b`) to a double-fault program. We use perfect (**1P**) to classify a singleton localisation, non-perfect (**1nP**) includes other repair sites with the one detected fault. Finally, failures (**F**) are where the localisation fails to return either fault location, which we did not experience with this data set.

The dual-fault programs, which are the same scale of source code as the previous data set with similar test suite sizes, are localised within the same time window. Double-faults localise with run-times in the same order of magnitude while twin-faults match the run-times of our earlier single-fault programs, when using our optimisations. The absolute and percentage ranked localisation performance also shows a similar characteristic to the single-fault data set where additional failing test cases do not significantly change the performance of the tool. However, the added complexity of this search does not provide identical narrowing performance. The whitelist localisation results exhibit some variability. This may be caused by the randomised test case order influencing the search space exploration. Twin-faults that find additional locations for potential repairs when exercised by larger test suites (2P to 2nP shift) may also indicate the limitations of smaller test suites to reason over which locations can be dropped as potential repairs. However, the unvarying ranked performance indicates the locations used by students to repair the submissions are not penalised by this limited data leading to a tighter localisation with the original test suites.

This tool extension demonstrates localisation that works on programs beyond our original single-fault slice in Sect. 5.1, without the likelihood of exploding symbolic analyser time costs caused by implementing a model-based dual-fault search using multiple toggle variables. We find this extension allows the processing of many additional student programs that occur in our database, extending the use of this tool beyond the single-fault programs previously localised. Programs with twin-faults are often perfectly localised without requiring test suites be classified (by if they might traverse any two assignments in a single trace) by a preprocessing stage or knowing in advance if the locations being searched are in a twin-fault relationship.

5.8 Grading support

To confirm the value of this localisation data for detecting "almost-correct" student submissions that test suites do not highlight, we extracted the test suite results for the data set. As each of these student submissions was selected because there exists a single edit to an assignment that brings it into compliance with the complete instructor-authored test suite, they should be graded within a generally narrow distribution, reasonably close to a fully compliant solution. However, as Fig. 16 shows in a histogram of the test suite compliance score of these programs, this is not the case. We use 10% buckets starting with $<10\%$ and a final bucket just for 100% compliance. The average submission fails for 51% of the test cases (dashed line) and the distribution of scores is very uneven, not clustering around a single value. As there are so few test cases per submission, the histogram buckets have been set to best show the distribution curve.

When expanding out to the much larger test suite in Fig. 17, the average submission fails for 59% of the test cases (dashed line). Here the distribution of grades is less flat, with

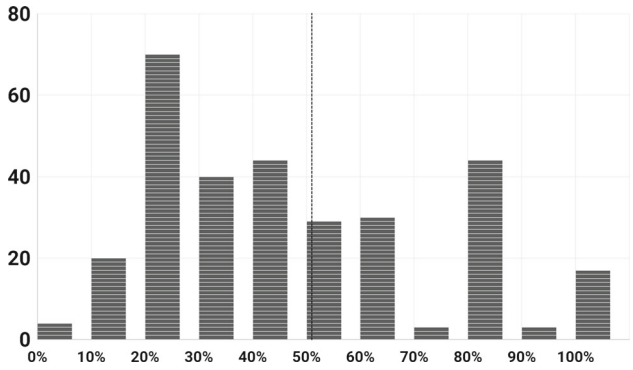


Fig. 16 Histogram of test suite compliance score for "almost-correct" programs using the original test suite

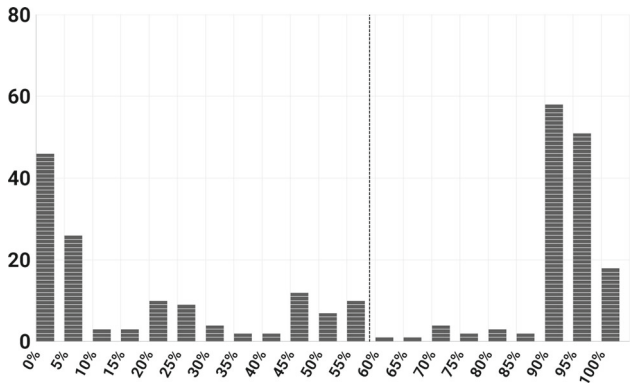


Fig. 17 Histogram of test suite compliance score for "almost-correct" programs using the extended test suite

clumping at both poles. We show this granularity by moving to 5% buckets, again including a final bucket just for 100% compliance. Half of the "almost-correct" submissions are scored with 80% or more of the test suite failing. Our tool will accelerate automation-assisted marking by flagging nearly-good code with the likely location of a repair that will radically increase a test suite-based grading.

5.9 Threats to validity

Comparing data for submissions originally written in Java to those in Python showed no significant skew to the data points explored. Translating a simple subset of each language into C syntax generated comparable sets. At least on this simple translated subset, the language originally used does not appear to strongly bias the characteristics of single faults introduced by students.

Our tool and the slice used on the database of student programs to generate the pairs for analysis makes a single-fault assumption. This is a common assumption in the fault localisation field [12,23,34] but does not reflect real-world debugging, although the existence of many code pairs in this data set does confirm real-world applicability. We have partially relaxed this constraint to a limited dual-fault assumption in Sect. 5.6.

The slicing of the student programs via a simple language translation stage, without any translation of library calls, and rejection of unparseable code restricts the form of programs explored. This slice assumes a repair possible via assignment modification. The choice of symbolic analyser (with an integer solver) also restricts the type of programs processed. Some programs are not suitable for automated localisation, such as those that never terminate on some inputs, and these would not make it through the database slice. These restrictions could add a bias to the student programs explored that could unfairly advantage one tool or call the generalisability of these results into question.

The open-source script used to execute the spectrum-based localisation was written in Java. Executing Java programs is known to come with a high initialisation time cost to start the JVM. Due to the short run-times involved, this may have inflated the run-time costs of this technique beyond some competing implementations based on the same GCov underlying tool.

To minimise the risk of over-tuning our design to the test data, our choice of time-out, sleep delay, and early termination values have not been tuned or selected for optimising with respect to the data set as this would compromise the generalisability of our results.

6 Related work

6.1 Fault Localisation

Localisation by examining counterexample traces, test cases, or other output from static and dynamic analysis tools is an active area of research [12,14,23,25,26,34,55,56,58].

Griesmayer et al. [23] have first applied model-based diagnosis methods to software. Our work follows the same lines; see Sect. 2 for a more detailed discussion. Griesmayer et al. [24] improve the original implementation to achieve times roughly comparable to our own initial re-implementation (see Sect. 4 for details). They also expand the possible fault locations to non-assignments (e.g. expressions in control flow guards), which could easily be applied to our approach as well, although the higher number of locations considered can lead to more complicated solver problems and thus higher run-times.

Königshofer and Bloem [36,37] have developed the FoREnSiC system, which includes a Griesmayer-style localisation. They have applied this to TCAS as well, but published results only for a few variants; here localisation times are more than an order magnitude slower (around 120 s) than our results. Königshofer et al. [38] report slightly improved

times (around 37 s) but had to annotate all functions with contracts, and so do no longer work from test suites alone.

Griesmayer's approach has also been applied to hardware designs in SystemC [40], often combined with different solver technologies such as QBF [62] or unsatisfiable cores [64]. Our results indicate that "plain old SAT/SMT" is still sufficient, but these technologies could be considered as alternatives in our approach as well.

Jose and Majumdar [34] convert an input C program to a maximum Boolean satisfiability problem that is analysed with MAX-SAT solver. However, because it returns the complement of the maximal subset of clauses that can be true for each single test case, their approach can omit genuine repair locations. It therefore relies on summing the results of the different test cases, providing a ranking of most to least commonly flagged locations. Thus the approach inherits the strengths/weaknesses of many heuristic-based fault localisation techniques. As discussed in Sect. 4, our approach provides comparable localisation times but a higher precision.

Spectrum-based fault localisation techniques, compared in [45,69,71], operate by examining passing and failing test cases separately. They assume that faults are more likely to be exercised by failing test cases and less likely to be exercised by passing test cases. The statements in a program can then be ranked based on the different weighting techniques. The analysis of the performance of these approaches is typically based on several scoring formulas that roughly correspond to how much of a program must be explored, given an ordered list of locations as tool output, before the genuine fault is found. The best-known example of this technique is the Tarantula tool [58] with TCAS results provided earlier [33]. Tarantula provides over 50% of the various variants of the small Siemens programs (including TCAS) with a localisation performance that ranks the injected fault location in the top 10% of lines in order of suspiciousness. But this ranking performance is inconsistent with 7% of these variants ranked such that the injected fault location is not in the top 80% of ranked locations. This requires debuggers to explore virtually the entire program before encountering the seeded fault when working through the program by suspiciousness value.

To counteract the single fault assumption limitation, it has been proposed that test cases can be grouped into clusters, each of which related to a different fault [69]. This allows Tarantula or similar tools to be run against all passing test cases and only the failing test cases which highlight a single fault, providing higher precision. Several different methods have been proposed to provide accurate clustering [69, p. 726]. As we discussed in Sect. 5.6, we consider that when using small programs, especially when processing many of them for MOOC scaling, the time cost of clustering will be a significant percentage of the total processing time.

State-of-the-art spectrum-based fault localisation methods have recently been compared using different theoretical frameworks [45,71]. Several methods, under these frameworks, have been identified that are maximal. Xie et al. expanded upon previously discovered formulas to generate several new maximal and distinct weightings using genetic programming [72]. The frameworks in these papers include the proof that these competing formulas cannot be placed into a hierarchy where one is strictly best for all inputs.

There are also empirical results over some of the Siemens small localisation benchmark programs. In Naish et al. (see Table XI, p. 11:23), the only methods identified as maximal under that paper's framework ranked the injected fault location on average at the 17th returned location (9.9%) over all TCAS variants. Tarantula returned the injected fault location at an average location between the 18th and 19th ranked location (10.8%).

This is significantly below the worst performance of the symbolic model checking approaches detailed in Table 2. The spectrum-based reporting metrics provide the average rank, as a percentage, in a ranked list of all lines of code. The symbolic model checking results report the total unranked lines flagged as suspicious, as a percentage of total lines of code. To compare these results, we must convert the unranked sets in Table 2 to ranked lists from which to derive averages. Randomly ranking all the returned lines above a list that randomly ranks all the lines not returned provides this conversion. The injected fault location, when it is returned as part of the unranked set, will, on average, be in the middle of the ranked returned lines. Using this conversion, the KLEE average result (4.6%) over the TCAS single-fault variants is equivalent to returning the injected fault location at the 4th ranked location (2.3%). As noted in Sect. 4.4, the different localisation scopes involved with each technique mean these results are not directly comparable. A spectrum-based approach will not only localise to assignment locations and TCAS is not ideally suited to providing these approaches with easily differentiable statements.

Delta Debugging [73] is a family of approaches that involve splitting up a large set of changes to find the minimal set that flip the program behaviour from correctly functioning to exhibiting a failure. This has variously been used to minimise inputs and traces but was later extended to source code exploration. The principle applied here [14] is to look at passing and failing traces and minimise the differences between them to isolate the failing components. This is reminiscent of a binary search, looking for interesting subset behaviour to narrow down variables that correlate with failure. However, this does require the existence of at least one passing trace and the localisation performance of Delta Debugging on the small Siemens programs [14] is worse than Tarantula's [33] results.

6.2 Student programming

Novice programmers can be divided into three classes [49, 68]: *stoppers*, who give up; *tinkerers*, who appear to modify their code at random; and *movers*, who are already able to engage with feedback to make progress. Even though the movers already demonstrate debugging skills beyond the average of their cohort, all three classes still need high-quality, novice-friendly debugging feedback to allow a self-guided refinement of their solutions towards a correct answer. Enhanced (syntax) error messages appear to be ineffectual in this respect [19], but when students are given hints regarding failures over a test suite, their effort increases compared to students not given hints [9]. On-demand programming feedback provides students the motivation to iterate on submissions [43]. Moreover, a correlation has been found between sessions where students were provided with feedback from a test suite and sessions that improved the student's score [61], indicating this assisted self-training. However, despite its benefits, students are somewhat resistant to a pure test-driven development (TDD) approach [8], as for example embodied in the Web-CAT [21] submission system, partly because TDD also requires expertise in developing test suites as a prerequisite to demonstrating programming ability.

A significant part of the debugging process is finding the location where the file needs to be changed to repair the fault [67]. Tools like AskIgor [74] provide cause chains to assist programmers in localising faults from test failures. While this lowers the threshold, it still requires too much debugging expertise for novices. Our tool instead provides a list of locations where a single assignment fault can be repaired to bring the program into compliance with the entire test suite. Giving tinkerers viable repair locations massively reduces the mutation space they are exploring. Giving movers viable repair locations directs their effort, allowing faster debugging times [48]. This should increase the chance that both will realise the repair and convert a program failure into a learning event.

For programmers to retain their status of movers, they must be provided with feedback in terms of suitable scaffolding to overcome progress stalls and achieve an otherwise unattainable goal [70]. Tinkerers who are unable to be scaffolded into movers will eventually become stoppers due to lack of progress [68]. Since the absolute number of locations shown before the actual repair site is critical to allowing progress before programmer interest drop-off generates stoppers [48], the provided list of locations to investigate must be short. Unfortunately, widespread spectrum-based fault localisation methods are unable to provide sufficiently short candidate lists. Pham et al. [51] tested Tarantula and Ochiai tools and on three variants of an algorithm, with an average length of 13 statements, they found over 9 statements shared the same top suspiciousness ranking. Localisation thus becomes a function of luck.

However, the feedback cannot be too prescriptive or too detailed. Complete program synthesis repair tools such as Auto Grader [59] remove all debugging or repair self-training from the process, trivialising the contribution and so learning of the student. Our tool bridges the gap between standard error reports, which many students lack the expertise to use, and fully automated repair suites.

Automated submission and assessment systems deliver immediate feedback to students for self-training, which is enjoyed by students and feeds into improvements throughout courses that use it [39]. They include tools to help accelerate grading of final submissions, providing both self-training and grading benefits [63]. Such assessment tools take many forms. Ceilidh [4] uses regular expressions to specify the test output of compiled student code, providing more freedom to define the expected answers than traditional test suites. ASSYST [31] contains style and complexity analysis tools that provide metrics beyond test suite correctness, and evaluate non-functional aspects such as code quality and efficiency. FrenchPress [7] focuses on analysis of code style and design, attempting to isolate errors only made by novices in Java programming. AutoGrader [28] requires students to program to a public interface, allowing whitebox testing at the cost of restricting the form of the code submissions. Pex4Fun [65] uses automated test case generation to guide student submissions and to scale to MOOC capacities. This approach requires students to demonstrate advanced debugging skills. Scheme-Robo [57] compares the program source to a model answer, assuming that Scheme's functional design limits the variability of meaningfully different programs that correctly answer the question.

Student code exhibits a wide range of program forms and performance characteristics [41,44,53,60] and students do not necessarily work towards solutions that are similar to an instructor-authored model answer. It is thus inadvisable to only provide feedback directing students towards model answers. Vujošević-Janičić et al. [32] use a static verifier as well as test suites when student solutions do not match the control flow of the model answer. A weighting system then mixes these feedback components to guide students to repair code errors and transform the structure towards that of the model answer or to grade the submission. ASys [30] attempts to overcome the limitation of functional testing using test suites with a customisable semi-automated grading system. Marking templates describe extensions to the test suite for a question, generating new tests specific to the student submission that verify each assessment property defined in the template. In their use on campus, 48% of the grading work was automatically handled by the tools. Interestingly, most of the marks difference between the group using ASys and the control group going through traditional grading was

accountable to errors made in the manual marking process. Similarly, our tool provides added feedback beyond functional grading using test suites, identifying student programs that "almost" conform to the question requirements and can be fixed with a single assignment edit. Our tool can also enrich existing systems like ASys, providing this alternative test of closeness to compliance when using test suites in other contexts.

Instructors spending several minutes per student per submission for feedback [42] cannot scale to MOOC environments with thousands of students per course. To test the scaling of an automated hint system to these environments, AutoTeach [2] provides students with access to a hint system that uncovers partial model answers to assist with learning. Such systems, while scaling very well, strictly constrain students to working towards a single model answer. However, the deployment of test suite grading for MOOCs is gaining traction [52] with the drawbacks of formatting issues (3.2% of submissions were awarded zero marks due to formatting issues, not functional failures) and failure to detect almost-correct submissions weighed against the benefits of providing some form of code assessment, as manual grading is not viable. Such issues can be reduced via importing more advanced tools into the grading system, assuming their run-time cost can be kept low enough for MOOC execution.

7 Conclusions and future work

Our main contribution in this article is an improved search algorithm through the test suite, reducing the effort for the symbolic execution of the model. Our results show Griesmayer's technique works in comparable time to the state of the art when driven with our optimised algorithm for the small C programs tested. This algorithm outperforms the naive reimplementation of the technique and the technique's originally published implementation by more than two orders of magnitude.

We generate genuine lists of repair locations as specified by test cases, for any repair that could be expressed as a lookup table for the right-hand side of an assignment, within the limits of symbolic analyser accuracy. These lookup table repairs provide assignment values to correct all failing test cases in the suite. Our time performance is in line with recent alternative model-based fault localisation techniques, but narrows the location set further without rejecting any genuine repair locations where faults can be fixed by changing a single assignment. This is more consistent than the localisation performance of other techniques and does so without compromising the narrowing extent, which might be done to avoid the false negatives shown in the competition.

Applying our tool in an educational context must meet the demands of novice programmers who are unlikely to be capable of advanced debugging techniques. Coursework and charrettes provide an opportunity for fast, accurate fault localisation to assist in educational institutions, which have already built up databases and workflows around test suite specified exercises. We have demonstrated a fast, model-based fault localisation tool on a collection of single-fault, real-world student submissions. The high quality localisation information reduces the search space for novice debuggers working down a list of potential repair locations when compared to the results from spectrum-based techniques. In over a third of the sampled failing student programs, our tool used the test suite to provide a localisation result that uniquely identified the location later used by the student to repair the program and rejected all other locations. This reinforces the qualitative difference between model-based fault localisation that reasons over a model of the student program to derive a list of feasible repair locations compared to a spectrum-based ranking process that infers suspiciousness of each program statement. These short, often exact singleton, lists of potential assignment repair locations can direct students to the site of improvement, assisting in the construction of a final submission that fully complies with the test suite. The run-time cost of our high-quality localisation matches that of fast, inaccurate spectrum-based fault localisation, even with large test suites, ensuring that our approach is viable even at the scale required by MOOCs.

We have demonstrated that submissions that are a single assignment edit away from full compliance with a test suite would not be graded predictably if scoring were based on compliance with the test suite only. This confirms the need for tools that can isolate such "almost-correct" student submissions.

We have also proposed an extension to our algorithm to account for collections including dual-fault programs with different test case coverage of those faults. We have explored the frequency of different classes of dual-fault programs in our data set of student submissions and demonstrated the preliminary application of our tool. This has been done using an extended algorithm without the solver cost of using a classic extension to n-faults that uses multiple toggle variables.

Future studies into this tool can survey the real-world performance of our tool on new data sets and expanding our tools to new program types (including new languages), fault classes, and symbolic analysis tools. The underlying methods and optimised algorithms developed and implemented into our current tools are programming language and symbolic analyser agnostic. This will allow future tests using other programming languages and beyond the restrictions (such as no floating-point symbolic analysis) of the currently selected downstream components. Such an extension would provide results for the limits of scalability of this approach when attempting a wider range of larger programs than the Siemens suite. An extended transformation system that extracted more

implicit assignments or modelled non-assignment points of repair would demonstrate localisation on a larger set of program locations.

Our current research points towards the study of tool-assisted learning in classroom environments and the effect on student progress when one group is provided with this high-quality localisation information in typical student programming tasks. Such studies can validate this feedback as valuable for novices, improving total debugging time and reducing the number of students who stop before completing a source code submission fully conforming to the provided test suite. The integration of this localisation feedback into student integrated development environments must be managed to maximise and quantify student comfort and views on the ease-of-use of this additional information. Future studies could be done with direct interaction tracking, surveys, or analysis of achievement changes when this tool is introduced to an existing course.

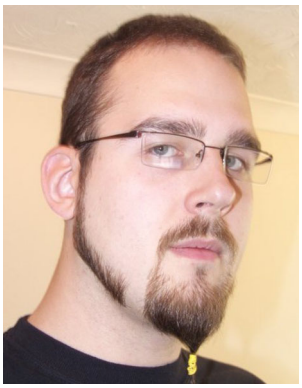
Acknowledgements We thank Ewan Tempero and Paul Denny at the University of Auckland for access to an anonymised copy of their student exercises and submissions to those exercises, from which we generated our student data set. We also thank Eli Bendersky, author of the PyCParser, the Automated Software Testing Group at Beihang University for Hawk-Eye, and contributors to all downstream components and tools used in our work. This academic research was funded by the Engineering and Physical Sciences Research Council, UK. Funding Number 1239954.

References

1. Ahmadzadeh, M., Elliman, D., Higgins, C.: An analysis of patterns of debugging among novice computer science students. In: Proceedings of the 10th Annual Conference on Innovation and Technology in Computer Science Education, (ITiCSE'05), pp. 84–88. ACM (2005)
2. Antonucci, P., Estler, C., Nikolić, D., Piccioni, M., Meyer, B.: An incremental hint system for automated programming assignments. In: Proceedings of the 20th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'15), pp. 320–325. ACM (2015)
3. Bendersky, E.: PyCParser C Parser and AST Generator Written in Python. http://github.com/eliben/pycparser (2012)
4. Benford, S.D., Burke, E.K., Foxley, E., Higgins, C.A.: The Ceilidh system for the automatic grading of students on programming courses. In: Proceedings of the 33rd Annual Southeast Regional Conference, ACM-SE 33, pp. 176–182. ACM (1995)
5. Birch, G., Fischer, B., Poppleton, M.: Fast model-based fault localisation with test suites. In: Proceedings of the 9th International Conference on Tests and Proofs, TAP'15, LNCS vol. 9154, pp. 38–57. Springer, Berlin (2015)
6. Birch, G., Fischer, B., Poppleton, M.: Using fast model-based fault localisation to aid students in self-guided program repair and to improve assessment. In: Proceedings of the 21st Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'16), pp. 168–173. ACM (2016)
7. Blau, H., Moss, J.E.B.: FrenchPress gives students automated feedback on java program flaws. In: Proceedings of the 20th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'15), pp. 15–20. ACM (2015)
8. Buffardi, K., Edwards, S.H.: Exploring influences on student adherence to test-driven development. In: Proceedings of the 17th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'12), pp. 105–110. ACM (2012)
9. Buffardi, K., Edwards, S.H.: Responses to adaptive feedback for software testing. In: Proceedings of the 19th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'14), pp. 165–170. ACM (2014)
10. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: Automatically generating inputs of death. In: Transactions on Information and System Security, vol. 12(2), p. 10A. ACM (2008)
11. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08), pp. 209–224. USENIX (2008)
12. Chandra, S., Torlak, E., Barman, S., Bodik, R.: Angelic debugging. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE'11), pp. 121–130. ACM (2011)
13. Cheeseman, P., Kanefsky, B., Taylor, W.M.: Where the really hard problems are. In: Proceedings of the 12th International Joint Conference on Artificial Intelligence, vol. 1, (IJCAI'91), pp. 331–337. Morgan Kaufmann (1991)
14. Cleve, H., Zeller, A.: Locating causes of program failures. In: Proceedings of the 27th International Conference on Software Engineering (ICSE'05), pp. 342–351. ACM (2005)
15. Console, L., Friedrich, G., Dupré, D.T.: Model-based diagnosis meets error diagnosis in logic programs. In: Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging (AADEBUG'93), pp. 85–87. Springer, Berlin (1993)
16. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the 3rd Annual Symposium on Theory of Computing (STOC'71), pp. 151–158. ACM (1971)
17. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. Trans. Softw. Eng. **38**(4), 957–974 (2012). IEEE
18. de Kleer, J., Williams, B.: Diagnosing multiple faults. Artif. Intell. **32**(1), 97–130 (1987). Elsevier
19. Denny, P., Reilly, A.L., Carpenter, D.: Enhancing syntax error messages appears ineffectual. In: Proceedings of the 19th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'14), pp. 273–278. ACM (2014)
20. Do, H., Elbaum, S., Rothermel, G.: Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. Empir. Softw. Eng. **10**(4), 405–435 (2005). Springer
21. Edwards, S.H.: Using software testing to move students from trial-and-error to reflection-in-action. In: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'04), pp. 26–30. ACM (2004)
22. GCC's GCov Source Code Coverage Analysis Tool. http://gcc.gnu.org/onlinedocs/gcc/Gcov.html
23. Griesmayer, A., Staber, S., Bloem, R.: Automated fault localization for C programs. In: Proceedings of the Workshop on Verification and Debugging (V&D'06), ENTCS, vol. 174(4), pp. 95–111. Elsevier (2007)

24. Griesmayer, A., Staber, S., Bloem, R.: Fault localization using a model checker. Softw. Test. Verif. Reliab. **20**(2), 149–173 (2010). Wiley
25. Groce, A., Chaki, S., Kroening, D., Strichman, O.: Error explanation with distance metrics. Int. J. Softw. Tools Technol. Transf. **8**(3), 229–247 (2006). Springer
26. Groce, A., Visser, W.: What went wrong: explaining counterexamples. In: Proceedings of the 10th International Conference on Model Checking Software (SPIN'03), pp. 121–136. Springer, Berlin (2003)
27. Hawk-Eye Statement Coverage-Based Tool for Automatic Fault Localization. http://code.google.com/archive/p/hawk-eye/ (2010)
28. Helmick, M.T.: Interface-based programming assignments and automatic grading of java programs. In: Proceedings of the 12th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'07), pp. 63–67. ACM (2007)
29. Hutchins, M., Foster, H., Goradia, T., Ostrand, T.: Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In: Proceedings of the 16th International Conference on Software Engineering (ICSE-16), pp. 191–200. IEEE (1994)
30. Insa, D., Silva, J.: Semi-automatic assessment of unrestrained java code: a library, a DSL, and a workbench to assess exams and exercises. In: Proceedings of the 20th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'15), pp. 39–44. ACM (2015)
31. Jackson, D., Usher, M.: Grading student programs using ASSYST. In: Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'97), pp. 335–339. ACM (1997)
32. Janičić, M.V., Nikolić, M., Tošić, D., Kuncak, V.: Software verification and graph similarity for automated evaluation of students' assignments. Inf. Softw. Technol. **55**(6), 1004–1016 (2013). Elsevier
33. Jones, J.A., Harrold, M.J.: Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th International Conference on Automated Software Engineering (ASE'05), pp. 273–282. ACM (2005)
34. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: Proceedings of the 32nd Conference on Programming Language Design and Implementation (PLDI'11), pp. 437–446. ACM (2011)
35. Joy, M., Griffiths, N., Boyatt, R.: The boss online submission and assessment system. J. Educ. Resour. Comput. **5**(3), 2A (2005). ACM
36. Könighofer, R., Bloem, R.: Automated error localization and correction for imperative programs. In: Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD'11), pp. 91–100. IEEE (2011)
37. Könighofer, R., Bloem, R.: Repair with on-the-fly program analysis. In: Proceedings of the 8th International Haifa Verification Conference (HVC'12), LNCS vol. 7857, pp. 56–71. Springer, Berlin (2012)
38. Könighofer, R., Toegl, R., Bloem, R.: Automatic error localization for software using deductive verification. In: Proceedings of the 10th International Haifa Verification Conference (HVC'14), LNCS, vol. 8855, pp. 92–98. Springer, Berlin (2014)
39. Laakso, M.J., Salakoski, T., Korhonen, A., Malmi, L.: Automatic assessment of exercises for algorithms and data structures—a case study with TRAKLA2. In: Proceedings of the 4th Finnish/Baltic Sea Conference on Computer Science Education, pp. 28–36. Uni. of Joensuu (2004)
40. Le, H.M., Grosse, D., Drechsler, R.: Automatic TLM fault localization for SystemC. Trans. Comput. Aided Des. Integr. Circuits Syst. **31**(8), 1249–1262 (2012). **IEEE**
41. Lister, R., Clear, T., Simon, Bouvier, D.J., Carter, P., Eckerdal, A., Jacková, J., Lopez, M., McCartney, R., Robbins, P., Seppälä, O., Thompson, E.: Naturally occurring data as research instrument: analyzing examination responses to study the novice programmer. In: SIGCSE Bulletin, vol. 41(4), pp. 156–173. ACM (2010)
42. MacWilliam, T., Malan, D.J.: Streamlining grading toward better feedback. In: Proceedings of the 18th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'13), pp. 147–152. ACM (2013)
43. Malmi, L., Korhonen, A., Saikkonen, R.: Experiences in automatic assessment on mass courses and issues for designing virtual courses. In: Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'02), pp. 55–59. ACM (2002)
44. McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.B.D., Laxer, C., Thomas, L., Utting, I., Wilusz, T.: A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In: Working Group Reports from 6th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE-WGR'01), pp. 125–180. ACM (2001)
45. Naish, L., Lee, H.J., Ramamohanarao, K.: A model for spectra-based software diagnosis. In: Trans. Softw. Eng. Methodol. **20**(3), 11A (2011)
46. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. Trans. Program. Lang. Syst. **1**(2), 245–257 (1979). ACM
47. Nguyen, H.D.T., Qi, D., Roychoudhury, A., Chandra, S.: SemFix: program repair via semantic analysis. In: Proceedings of the 35th International Conference on Software Engineering (ICSE'13), pp. 772–781. ACM (2013)
48. Parnin, C., Orso, A.: Are automated debugging techniques actually helping programmers? In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'11), pp. 199–209. ACM (2011)
49. Perkins, D.N., Hancock, C., Hobbs, R., Martin, F., Simmons, R.: Conditions of learning in novice programmers. J. Educ. Comput. Res. **2**(1), 37–55 (1986). Sage
50. Pham, H.: Software Reliability. Springer, Berlin (2000)
51. Pham, L.H., Trinh, G.V., Dinh, M.H., Mai, N.P., Quan, T.T., Ngo, H.Q.: Assisting students in finding bugs and their locations in programming solutions. Int. J. Qual. Assur. Eng. Technol. Educ. **3**(2), 12–27 (2014). IGI
52. Pieterse, V.: Automated assessment of programming assignments. In: Proceedings of the 3rd Computer Science Education Research Conference on Computer Science Education Research (CSERC'13), pp. 45–56. ACM (2013)
53. Reilly, A.L., Denny, P., Kirk, D., Tempero, E., Yu, S.Y.: On the differences between correct student solutions. In: Proceedings of the 18th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'13), pp. 177–182. ACM (2013)
54. Reiter, R.: A theory of diagnosis from first principles. Artif. Intell. **32**(1), 57–95 (1987). Elsevier
55. Renieres, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: Proceedings of the 18th International Conference on Automated Software Engineering (ASE'03), pp. 30–39. IEEE (2003)
56. Sahoo, S.K., Criswell, J., Geigle, C., Adve, V.: Using likely invariants for automated software fault localization. In: Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13), pp. 139–152. ACM (2013)
57. Saikkonen, R., Malmi, L., Korhonen, A.: Fully automatic assessment of programming exercises. In: Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'01), pp. 133–136. ACM (2001)
58. Santelices, R., Jones, J.A., Yu, Y., Harrold, M.J.: Lightweight fault-localization using multiple coverage types. In: Proceedings of the

31st International Conference on Software Engineering (ICSE'09), pp. 56–66. IEEE (2009)
59. Singh, R., Gulwani, S., Lezama, A.S.: Automated feedback generation for introductory programming assignments. In: Proceedings of the 34th Conference on Programming Language Design and Implementation (PLDI'13), pp. 15–26. ACM (2013)
60. Soloway, E., Spohrer, J.C.: Studying the Novice Programmer. Erlbaum, Hillsdale (1988)
61. Spacco, J., Fossati, D., Stamper, J., Rivers, K.: Towards improving programming habits to create better computer science course outcomes. In: Proceedings of the 18th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'13), pp. 243–248. ACM (2013)
62. Staber, S., Bloem, R.: Fault localization and correction with QBF. In: Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing (SAT'07), LNCS vol. 4501, pp. 355–368. Springer, Berlin (2007)
63. Striewe, M., Balz, M., Goedicke, M.: A flexible and modular software architecture for computer aided assessments and automated marking. In: Proceedings of the 1st International Conference on Computer Supported Education (CSEDU'09) pp. 54–61. INSTICC (2009)
64. Sülflow, A., Fey, G., Bloem, R., Drechsler, R.: Debugging design errors by using unsatisfiable cores. In: Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV'08), pp. 159–168. ITG/GI/GMM Workshop (2008)
65. Tillmann, N., De Halleux, J., Xie, T., Gulwani, S., Bishop, J.: Teaching and learning programming and software engineering via interactive gaming. In: Proceedings of the 35th International Conference on Software Engineering (ICSE'13), pp. 1117–1126. IEEE (2013)
66. Vessey, I.: Expertise in debugging computer programs: a process analysis. Int. J. Man Mach. Stud. **23**(5), 459–494 (1985). Academic Press
67. Wang, Q., Parnin, C., Orso, A.: Evaluating the usefulness of IR-based fault localization techniques. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'15), pp. 1–11. ACM (2015)
68. Whalley, J., Kasto, N.: A qualitative think-aloud study of novice programmers' code writing strategies. In: Proceedings of the 19th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'14), pp. 279–284. ACM (2014)
69. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. Trans. Softw. Eng. **42**(8), 707–740 (2016). IEEE
70. Wood, D., Bruner, J.S., Ross, G.: The role of tutoring in problem solving. J. Child Psychol. Psychiatry **17**(2), 89–100 (1976). Pergamon
71. Xie, X., Chen, T.Y., Kuo, F.C., Xu, B.: A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. Trans. Softw. Eng. Methodol. **22**(4), 31A (2013). ACM
72. Xie, X., Kuo, F.C., Chen, T.Y., Yoo, S., Harman, M.: Provably optimal and human-competitive results in SBSE for spectrum based fault localisation. In: Proceedings of the 5th International Symposium on Search Based Software Engineering (SSBSE'13), LNCS vol. 8084, pp. 224–238. Springer, Berin (2013)
73. Zeller, A.: Yesterday, my program worked. Today, it does not. Why? In: Proceedings of the 7th European Software Engineering Conference, ESEC/FSE-7, LNCS vol. 1687, pp. 253–267. Springer, Berlin (1999)
74. Zeller, A.: Isolating cause-effect chains with AskIgor. In: Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC'03), pp. 296–297. IEEE (2003)



Geoff Birch received their Ph.D. in Computer Science from the University of Southampton in 2016 on the topic of Fast, Fully-Automated, Model-Based Fault Localisation and Repair with Test Suites as Specification. Their research interests include metaprogramming and real-time rendering, with a current focus on applications of static and dynamic code analysis.

Bernd Fischer is professor for computer science at Stellenbosch University, South Africa. He received his Ph.D. degree from the University of Passau, Germany, and held positions at TU Braunschweig, NASA Ames Research Center, and University of Southampton before moving to South Africa. His research interests include code generation, programming languages, formal methods, and software verification, with a focus on bounded model checking.

Michael Poppleton received his B.Sc. in Mathematics and Mathematical Statistics from the University of the Witwatersrand in Johannesburg, South Africa in 1977. After a 13-year career in system and data communications development and testing, in 1991 he completed his M.Sc. at Aston University. In 2001 he completed his part-time Ph.D under supervision of Richard Banach at the University of Manchester. His thesis proposed the notion of retrenchment, a liberalization of the refinement method central to state-based Formal Methods. He then investigated application and development of retrenchment. At the University of Southampton, he used the Event-B language to investigate composition, decomposition, feature-based formal development, and integration with UML. Subsequent work examined formal and co-simulation modelling of Wireless Sensor Networks. Recent work includes the integration of temporal logic with the first-order Event-B for integrated safety/liveness reasoning in distributed systems.